

MACMETH

**A Fast Modula-2 Language System
For the Apple Macintosh**

User Manual

2.0
Institut für Informatik
ETH Zürich
August 1986



MACMETH

**A Fast Modula-2 Language System
For the Apple Macintosh**

User Manual

**2.0
Institut für Informatik
ETH Zürich
August 1986**

Disclaimer

The authors of the computer software described in this manual hereby disclaim any and all guarantees and warranties on the software or its documentation, both expressed or implied. No liability of any form shall be assumed by the authors. Any user of this software uses it at his or her own risk.

This product is sold on an "as is" basis; no fitness for any purpose whatsoever nor warranty of merchantability are claimed or implied.

The authors reserve the right to make changes, additions, and improvements to the software or documentation at any time without notice to any person or organization; no guarantee is made that future versions of either will be compatible with any other versions.

Authors

Niklaus Wirth
Jürg Gutknecht
Werner Heiz
Hansruedi Schär
Hermann Seiler
Christian Vetterli

Copyright

Copyright (c) 1986 by Institut für Informatik
Eidgenössische Technische Hochschule (ETH) Zürich
Clausiusstr. 55, CH-8092 Zürich, Switzerland

All rights reserved. Duplication of this work by any means is forbidden without prior written consent of the Institut für Informatik.

Distribution, Licences

Official distributors of MacMETH (software and documentation) are

Werner Heiz
Neugasse 71
CH-8005 Zürich
Switzerland

Modula Corporation
950 N. University Ave.
Provo, Utah 84604
USA

Apply for "Simple CPU", School, and Site Licences.

Contents

1. Introduction and Startup Guide	
1.1. Introduction	4
1.2. System Description	4
1.3. Getting Started	5
1.4. Keyboard and Mouse, Special Keys	8
1.5. References	8
2. The Program Environment	
2.1. Introduction	10
2.2. Starting MacMETH	11
2.3. Several Ways to Work	11
3. The Editor	
3.1. Introduction	13
3.2. Starting the Editor	13
3.3. Text Entry	14
3.4. Text Selection	15
3.5. Scrolling	15
3.6. Menu Commands	15
4. The Compiler	
4.1. Introduction	19
4.2. Starting the Compiler	19
4.3. Compilation	20
4.4. Program Execution	21
4.5. The Implemented Language	22
4.6. Modula-2 Syntax	27
4.7. Compiler Error Messages	29
5. The Debugger	
5.1. Introduction	33
5.2. Starting the Debugger	34
5.3. Global Commands	35
5.4. Local Commands	36
6. Utility Programs	
6.1. The Linker	37
6.2. The Decoder	39
7. Library Modules	
7.1. Introduction	41
7.2. InOut	42
7.3. RealInOut, LongRealInOut	44
7.4. MathLib, LongMathLib	45
7.5. FileSystem	46
7.6. Terminal, TerminalIn, TerminalOut	49
7.7. Printer	51
7.8. CursorMouse	52
7.9. Menu	54
7.10. Windows	55
7.11. TextWindows	57
7.12. GraphicWindows	59
7.13. EventBase	61
7.14. System	63
7.15. Accessing the Macintosh Toolbox	66

1. Introduction and Startup Guide

Werner Heiz, 12-Aug-86

1.1. Introduction

This manual describes the implementation of the Modula-2 development system *MacMETH*. It is neither a reference manual nor a course about programming in Modula-2 (refer to [1] for an introduction to the language).

MacMETH has been developed in the group Computer Systeme at the Institut für Informatik, ETH Zürich, in co-operation with the Rechenzentrum, ETH Zürich. The goal was to have a tool for teaching Modula-2 on our Macintosh computers. The following diagram gives a project overview:

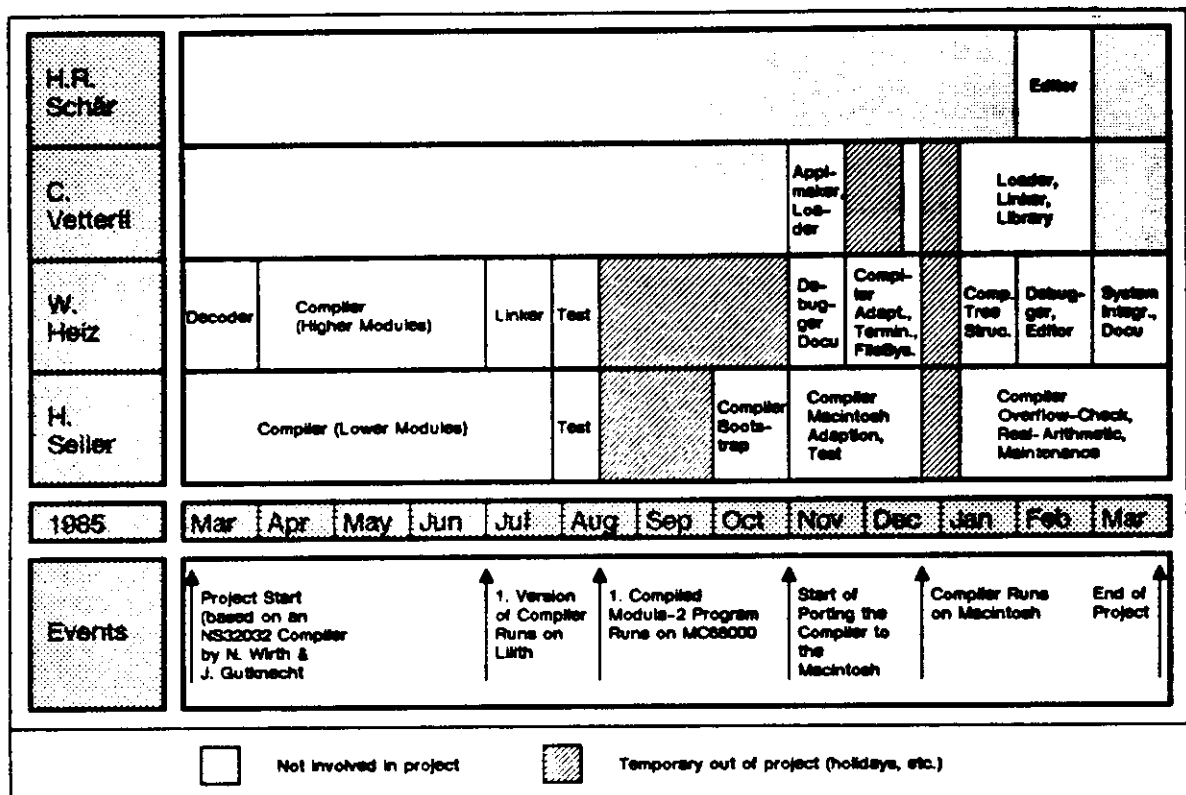


Fig. 1.1: Project History

1.2. System Description

System Features

The main features of the MacMETH system can be summarized as follows:

- Compiler:
- single-pass Modula-2 compiler
 - full language Modula-2 supported (restriction: declarations must precede references)
 - compiles at a speed of 1500 lines/minute on diskette, 3000 lines/minute on hard disk
 - generates native MC68000 code

- object code relocatable, no explicit linking necessary
 - Toolbox calls generate inline traps
 - each Modula-2 program can be converted into a Macintosh standalone application
- Debugger:**
- symbolic debugger with multiple windows
 - displays process and data state of an erroneous program
 - allows "zooming" into data structures
- Editor:**
- program editor, no restriction on file size
 - supports multiple files and windows
 - displays compiler detected errors
- MacMETH:**
- compiler and editor remain loaded (once they are activated)
 - very fast program switching
 - runs on one 400 kByte diskette without diskette-swapping
 - hierarchical file system (paths) supported
 - MacMETH compiles itself (everything is written in Modula-2)

System Requirements

The MacMETH system requires a 512 kByte Macintosh (*Fat Mac*). An external floppy drive is not necessary for small programs (a minimum configuration with system, editor, compiler, and debugger leaves about 50 kByte of work space on a 400 kByte diskette). For serious program development an external drive (or even a hard disk) is recommended.

MacMETH fully supports the hierarchical file system of the Macintosh Plus by allowing the user to specify alternate search paths in his own configuration file (i.e. "User.Profile").

1.3. Getting Started

The first thing to do is to make a copy of the distribution diskette. From now on, work with the copy and put the original diskette in a safe place.

Contents of the Distribution Diskette (not bootable!)

MacMETH	Main application (calls editor, etc.)
Edit	Program editor (pre-linked)
Compile	Modula-2 compiler (pre-linked)
Debug	Debugger (pre-linked)
Link	Application maker/linker
User.Profile	User defined search paths
ErrList.DOK	List of compiler error messages
CursorMouse.SBM	Mouse handling and cursor tracking (.SBM = symbol file, .OBM = object file)
CursorMouse.OBM	
EventBase.SBM	Simple event scheduler
EventBase.OBM	
FileSystem.SBM	Basic file input/output procedures
FileSystem.OBM	
GraphicWindows.SBM	Graphic window handling
GraphicWindows.OBM	

InOut.SBM	Simple handling of formatted input/output
InOut.OBM	
LongMathLib.SBM	Mathematical functions for LONGREAL
LongMathLib.OBM	
LongRealInOut.SBM	Input/output of LONGREAL numbers
LongRealInOut.OBM	
MathLib.SBM	Mathematical functions for REAL
MathLib.OBM	
Menu.SBM	Command selection from a menu
Menu.OBM	
Printer.SBM	Basic printer output procedures
Printer.OBM	
RealInOut.SBM	Input/output of REAL numbers
RealInOut.OBM	
RIO.OBM	Used by Debug/RealInOut/LongRealInOut
System.SBM	Heart of MacMETH
Terminal.SBM	Basic input/output procedures
Terminal.OBM	
TerminalIn.SBM	Basic input procedures
TerminalIn.OBM	
TerminalOut.SBM	Basic output procedures
TerminalOut.OBM	
TextWindows.SBM	Text window handling
TextWindows.OBM	
Windows.SBM	Window handling (contents irrelevant)
Windows.OBM	
Buggy.MOD	Compile this first! (system demo)
Example.MOD	Debugger demo
Eratos.MOD	Benchmark <i>Sieve of Eratosthenes</i>
Loader.MOD	Allows working at the <i>Finder</i> level
dec68k.MOD	Object file decoder
Rectangles.MOD	Simple graphic demo program
QuickDraw.DEF	Demonstrates QuickDraw access
QuickDraw.MOD	
FileMgr.DEF	Demonstrates operating system access
FileMgr.MOD	

Minimum System Configuration

If you want to work with one drive only and don't like swapping diskettes, prepare a diskette (must NOT be write-protected) with the following files on it:

Macintosh System & Finder (small version)
 MacMETH
 Edit
 Compile
 Debug (when needed)
 User.Profile
 ErrList.DOK

Add Library modules as needed by your program. Remember that MacMETH requires the

library modules CursorMouse.OBM, EventBase.OBM, FileSystem.OBM, Menu.OBM, Printer.OBM, RIO.OBM, TerminalIn.OBM, TextWindows.OBM, and Windows.OBM. Now, you should have about 30 kByte of free room on your 400 kByte diskette.

Working with multiple Drives, Diskettes & Directories

Macintosh file names have the following syntactical structure:

FileName = Path Name.
 Path = [VolName ":"] {SubDirName ":" | ":" }.

VolName is the volume name (each Macintosh diskette has a name).

SubDirName is the name of a subdirectory/folder.

Name is the local name of the file.

Examples of correct Macintosh file names:

MacMETH 2.0:File1	disk name followed by local file name
Untitled:Library:TextWindows.DEF	disk name, folder name, local file name
:Library:RealInOut.MOD	relative path starting at current folder
::User2:Temp	relative path starting one folder above current folder

Normally, you only want to deal with local file names and forget about paths. As folders are a useful means for organizing files, a mechanism in MacMETH helps you localizing them. There is a configuration file (i.e. "User.Profile") where you give an ordered list of paths that will in turn be added to your file until it is found (if it exists in one of the given locations or in the current directory of the current disk which is searched first). The configuration file must lie in the same folder as the application MacMETH, it is read only once (at the start up process). The output files of the MacMETH applications (i.e. Edit, Compile, Link, etc.) go to the same location where the input file comes from.

Here is a sample "User.Profile":

```
"PATH"
Library:, MacMETH 2.0:, Work:, Untitled:, :Library:, :Modula:
```

The first 4 path entries are disk names, the last 2 are relative folder names (it is advisable to use relative paths, as this keeps your file names small; some applications limit the maximum file name length to 32 characters). There is no limit on the path list length in the "User.Profile".

A Sample Edit/Compile/Run/Debug Session

The numbers in the text correspond to the numbers in the graphic (Fig1.2).

1. Start the system by double-clicking the MacMETH application. Select *Compile* from the *File* menu.
2. The compiler requests a file name. Enter "Buggy.MOD" and press <RETURN>. The compiler detects errors in the program.
3. Leave the compiler by typing <RETURN> and select *Edit* from the *File* menu. The editor suggests the last compiled file "Buggy.MOD" for editing.
4. Press <RETURN> to accept the default file name. The editor points with the caret to the erroneous position and displays the error message in the lower window. Correct the error, leave the editor and enter the compiler again. The compiler suggests "Buggy.MOD" for compiling.
5. Press <RETURN> to accept the default file name. The compiler generates 112 bytes of native MC68000 code in one pass. Leave the compiler and select *Execute* from the *File* menu.
6. Select *Buggy.OBM* from the file box using the mouse.
7. A runtime error occurs. Select *Debug* using the mouse.
8. The debugger displays five windows with source text (error position marked), procedure call chain, module list and local/global variables with name, type and value.

This session takes 60 seconds.

1.4. Keyboard and Mouse, Special Keys

The following notation is used throughout this documentation.

ML	mouse button
MM	command key (middle mouse button on the Lilith computer)
MR	option key (right mouse button on the Lilith computer)
EOL	return key
ESC	enter key
TAB	tabulator key
SPC	space bar
DEL	"←" key

For non-ASCII-keyboard character sets see also page 14.

1.5. References

- [1] **Programming in Modula-2**
N. Wirth, 3rd. Edition, Springer-Verlag, Heidelberg, New York, 1985.
- [2] **Inside Macintosh**
Apple Computer Inc., 467 Saratoga Avenue Suite 621, San Jose, CA 95129.

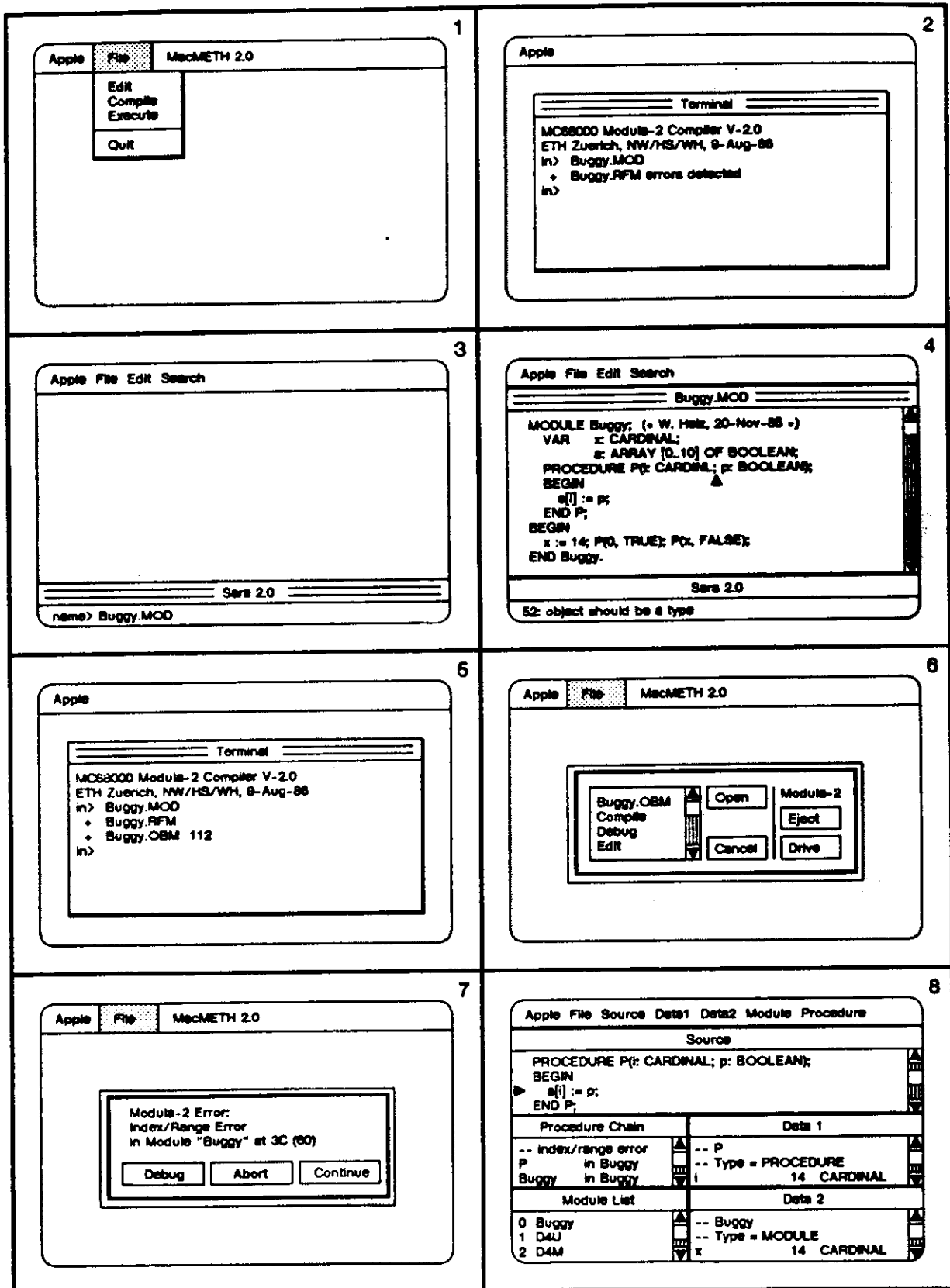


Fig. 1.2: A Sample Edit/Compile/Run/Debug Session

2. The Program Environment

Werner Heiz, 12- Aug-86

2.1. Introduction

The MacMETH system mainly consists of an editor, a compiler, and a debugger. On the distribution diskette they are stored in form of pre-linked Modula-2 programs and are invoked by the help of a shell (the Macintosh application MacMETH). The shell contains many often needed library modules as well as a loader for Modula-2 object files. The following picture gives a model of the MacMETH system on the Macintosh.

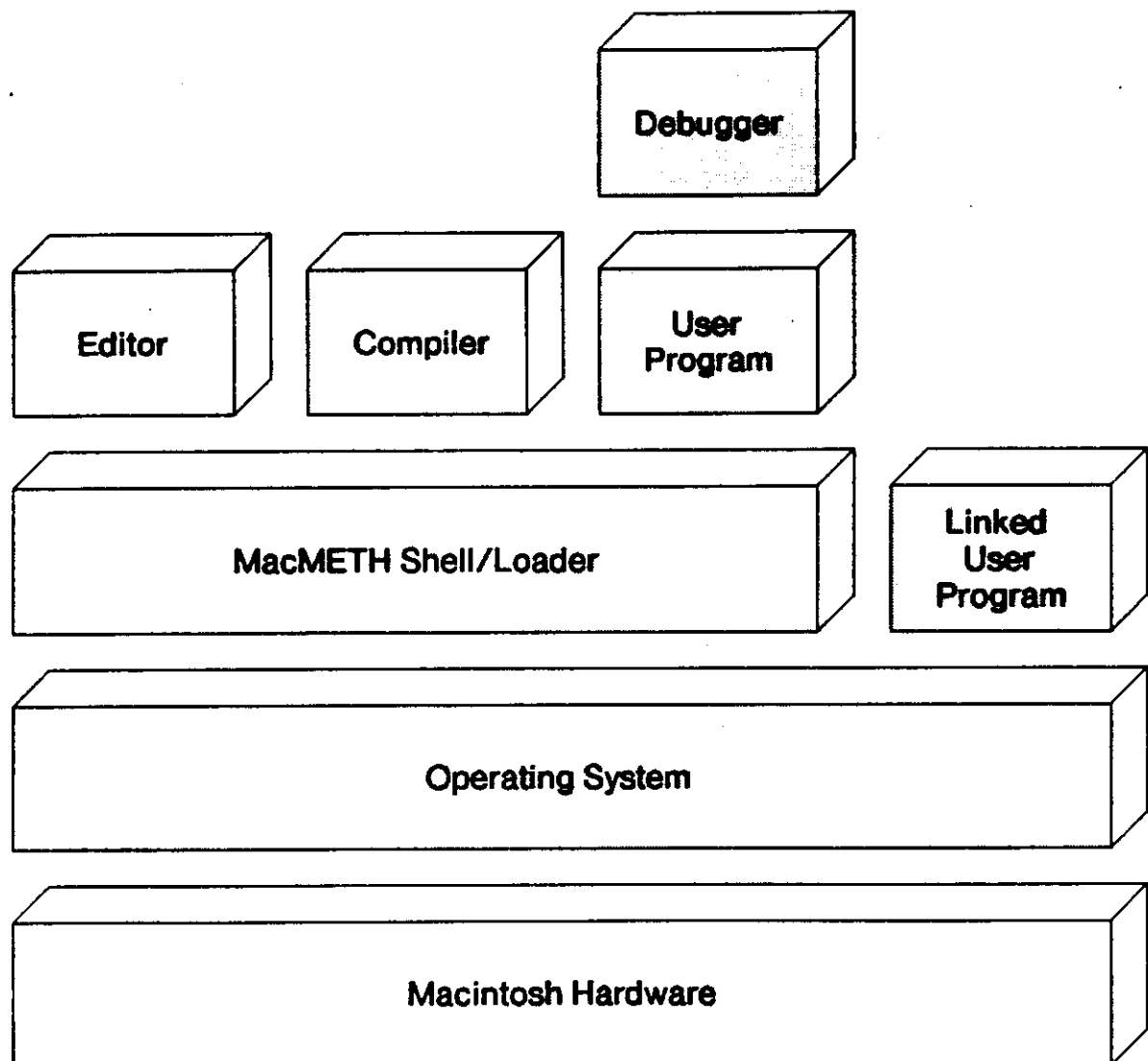


Fig. 2.1: MacMETH Layers

Each layer builds upon the resources of the next lower layer. The debugger is only called in case of a runtime error. MacMETH allows programs to dynamically call other programs (no linking necessary).

2.2. Starting MacMETH

You enter the shell by double-clicking the Macintosh application MacMETH at the *Finder* level. Then, the following screen image is presented:

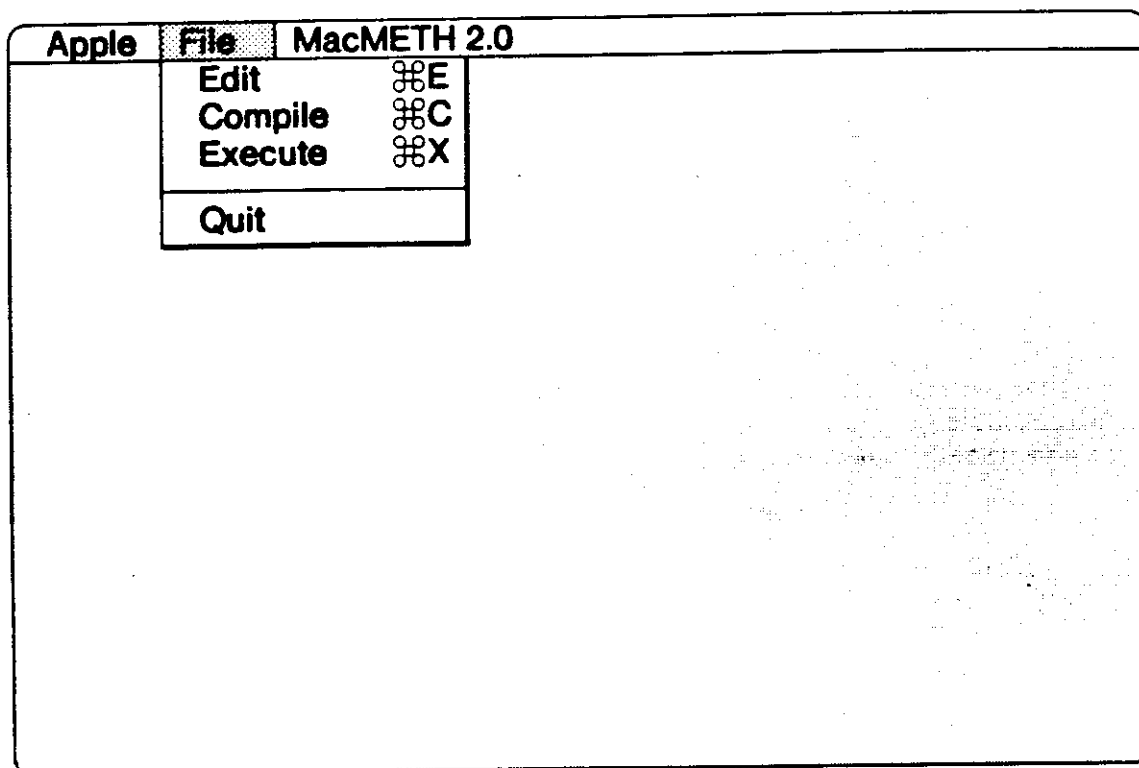


Fig. 2.2: Starting MacMETH

You have four options in the *File* menu: *Edit*, *Compile*, *Execute*, and *Quit* (the debugger is only called in case of a runtime error).

2.3. Several Ways to Work

There are several possible ways to work with the Modula-2 system depending on your preferences. Let's group the MacMETH users into three categories and see how they work with the system:

1. The Modula-2 program developer who likes to work on the fast MacMETH shell (most users will prefer that)
2. The Modula-2 program developer who wants to work at the *Finder* level (having his own favorite editor, perhaps)
3. The user of an application developed by MacMETH

Group 1

This is the fastest way to work with the Modula-2 system. You may enter the editor or the compiler through the *File* menu, and once they are loaded from the disk, they remain loaded as long as the shell is active. Please note, however, that the files you work with are always

saved on disk.

If you select *Execute* from the MacMETH *File* menu, you get the following screen image:

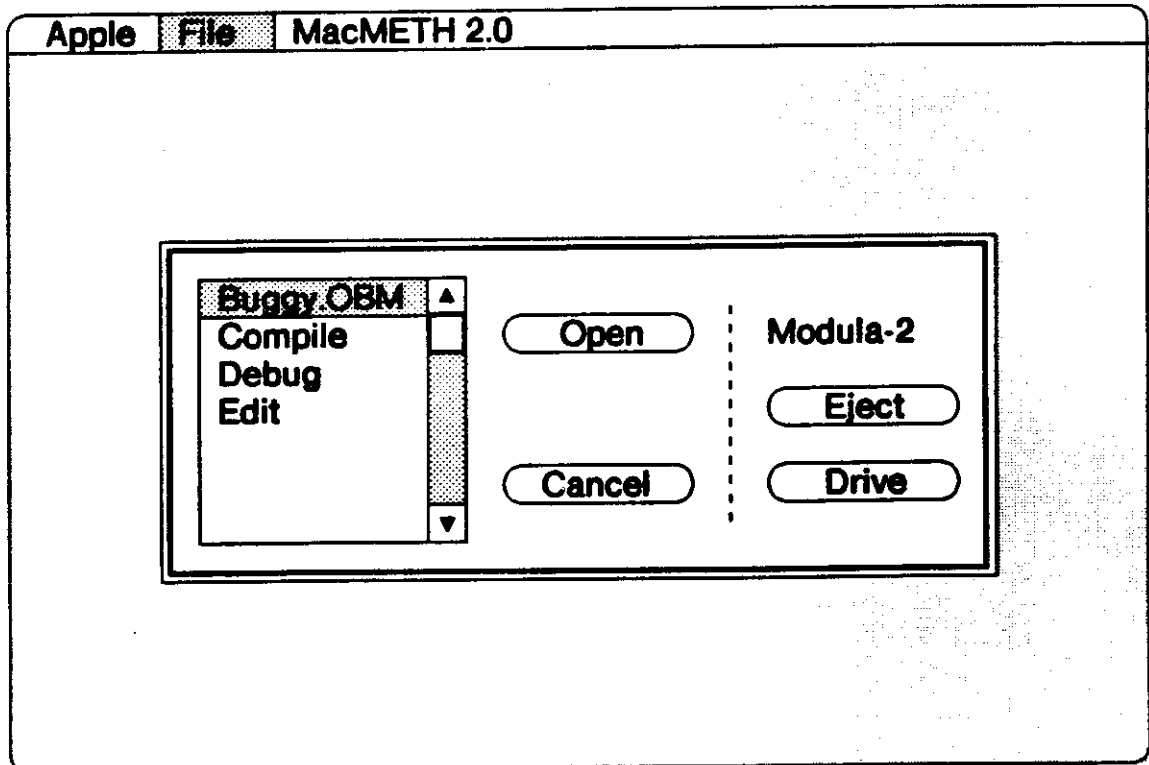


Fig. 2.3: Executing Programs

You can now select the program you want to execute and start it (all missing modules are loaded automatically). In particular, you can start also the editor and the compiler through this command. The difference, however, is that programs activated in this way do not remain loaded after termination.

Group 2

Every Modula-2 program can be transformed into a standalone Macintosh application. The editor, compiler, and debugger *are* Modula-2 programs. So, there can be no objection to convert them (by the help of the linker, see chapter 6) into standalone applications and work at the *Finder* level.

Group 3

Every Modula-2 program can be converted into a standalone Macintosh application, so the user doesn't have to know anything about MacMETH.

3. The Editor

Hansrudolf Schär, Werner Heiz, 12-Aug-86

3.1. Introduction

This chapter describes the use of the editor. It is a full-screen editor, specially designed to create and change programs (it uses a special interface to the compiler to display possible compilation errors). This editor imposes no restriction on file size, as it uses an incremental method (piece-list) for editing.

3.2. Starting the Editor

You start the editor by selecting *Edit* from the MacMETH-menu:

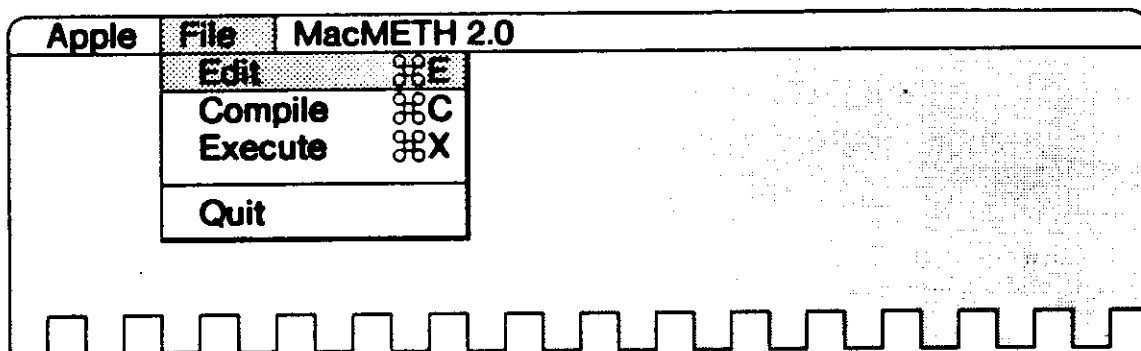


Fig. 3.1: Starting the Editor

Now the Macintosh presents the following screen image:

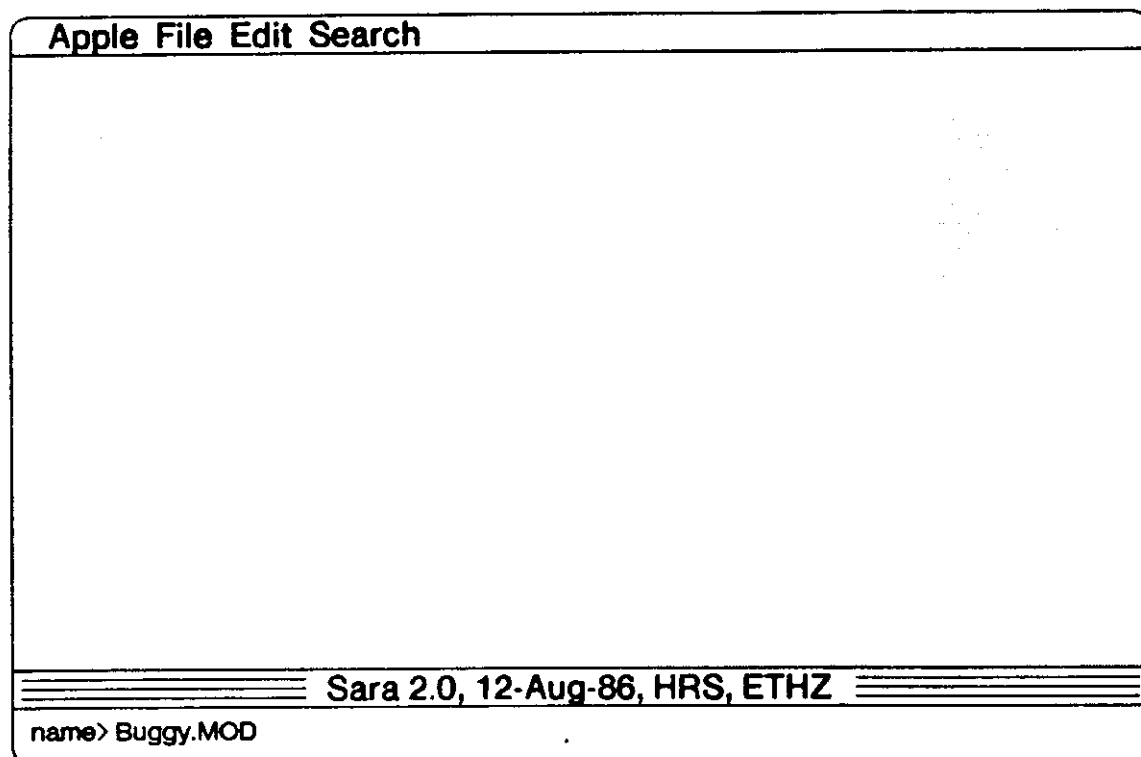


Fig. 3.2: Editor Initial Display

You see a small window on the bottom of the screen, possibly giving you a default file name for editing (the last compiled file is the default). Press <return>, <space> or the mouse button to accept the name, or enter an alternate name (to create a new text enter the empty name). If you end your file name with ".", the extension "MOD" is appended.

The editor now creates another window displaying the first page of your program. If your program contains errors (detected by the compiler in an earlier compilation) the caret (the text insertion point) is automatically placed after the first error. The dialog window displays the type of error.

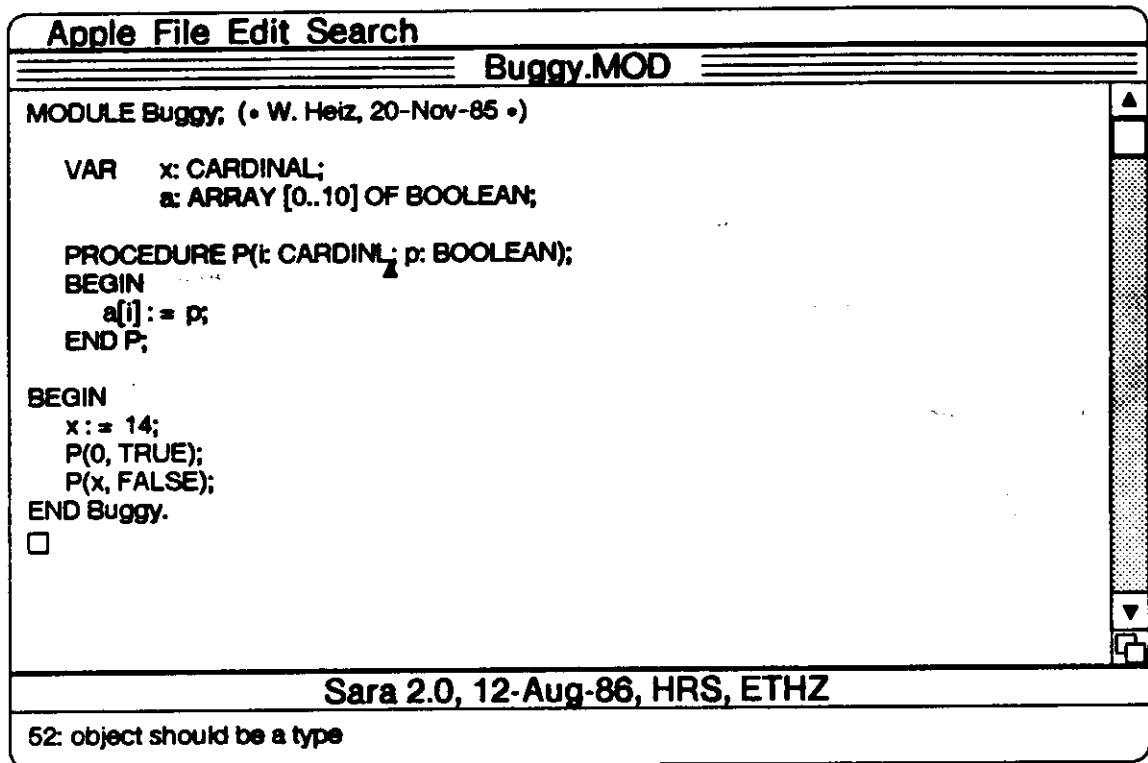


Fig. 3.3: Editor Screen Image

3.3. Text Entry

The text insertion point (caret) is represented by a small triangle which can be positioned by ML (press and release at the same place). The entered text is inserted at the caret and does not overwrite a previous text. You can delete the last character by typing DEL.

On the German keyboard, it is somewhat difficult to locate characters needed in Modula-2. The following table helps you finding them:

[	<option> 5
]	<option> 6
	<option> 7
{	<option> 8
}	<option> 9
↑	^^ or <option> <shift> 6 <option> <shift> 6
#	<option> <shift> 3

3.4. Text Selection

A text portion can be selected by moving the mouse over the text with the ML button down (dragging). If at the same time another button is active, the following commands are executed directly: *Copy* (MM), *Delete* (MR), or *Move* (MM and MR). The selection is canceled by pressing ESC. An existing text selection can be extended/shortened by pointing to the last selected character and dragging.

3.5. Scrolling

The editor uses the standard Macintosh scroll mechanism by the help of the scroll bar on the right.

3.6. Menu Commands

The editor commands can be classified into 3 groups: File, Edit, and Search.

- File open, split and print documents, quit.
- Edit copy, move, delete, adjust text, autoindentation on/off.
- Search find error, find string, replace.

The commands are explained in more detail below.

File

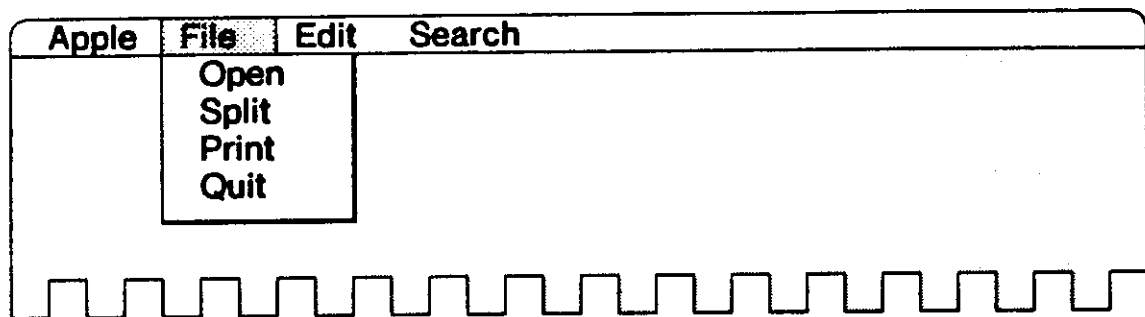


Fig. 3.4: File Commands

Open

Open a document. The editor allows you to open as many as 14 files at a time. The name is read from the keyboard (the default name is the name of the last compiled file or a valid text selection in the file). An empty name creates a new document.

name>

SPC, EOL, ML Terminate the input. A trailing "." is expanded to ".MOD".

DEL Delete the last character.

ESC, MM, MR Abort the command.

Split

Split window into subwindows. Each subwindow can be scrolled independently. Selection beyond subwindow boundaries is allowed (all text between is selected). This allows you to select text portions greater than the window size.

define separation line

ML, MM, MR Used to point to the location where separation should take place (dragging allowed).

ESC Abort the command.

Print

Print the file denoted by the caret.

printing ...

any key Pause the printing process.

ESC Abort the command.

Quit

Leave the editor without updating the document.

quit?

"y", "Y", ML Leave the editor.

any other key Abort the command.

Close

Documents and subwindows are closed by activating their *close boxes* (upper left square in title bar). The name is read from the keyboard (the default name is the name given in the *Open* command or a valid text selection). After closing the last file the *Quit* command is called automatically.

name>

SPC, EOL, ML Terminate the input. A trailing "." is expanded to ".MOD". The document is stored on disk and the previous version of the document gets the extension ".BAK".

DEL Delete the last character.

ESC, MM, MR Release the document without updating.

release?

"y", "Y", ML Close the document without updating.

any other key Abort the command.

Edit

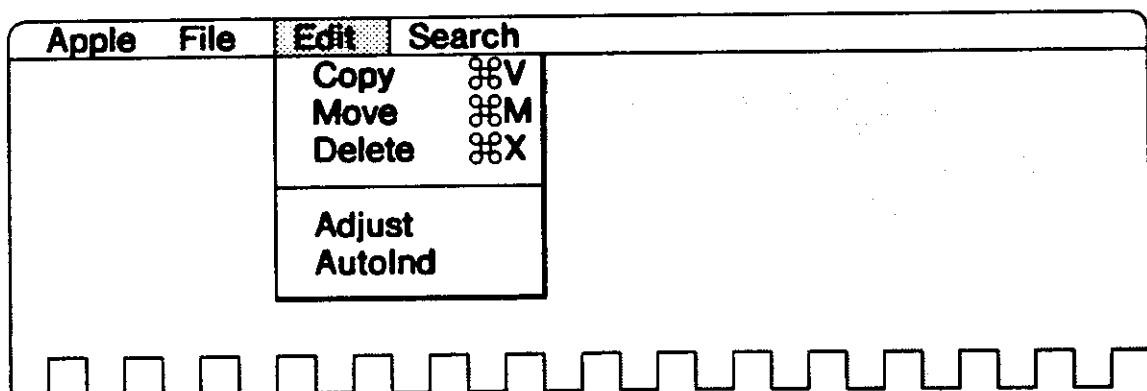


Fig. 3.5: Edit Commands

Copy

Copy a piece of text. If a text sequence is selected, it is copied to the internal buffer. Then the buffer contents are inserted at the caret.

Move

Move a piece of text. This command corresponds to the command sequence *Delete, Copy*.

Delete

Delete a piece of text. If a text sequence is selected, it is copied to the internal buffer and deleted.

Adjust

Shift a piece of text horizontally. The selected lines are shifted to the left or to the right (useful to indent a whole procedure).

adjust(-9..9)>

"0", "1", ..., "9" Shift 0, 1, ..., 9 position(s) to the right.

"-1", ..., "-9" Shift 1, ..., 9 position(s) to the left.

"<" Shift 1 position to the left.

">" Shift 1 position to the right.

"[", ML Shift 2 positions to the left.

"]", MR Shift 2 positions to the right.

"{" Shift 4 positions to the left.

"}" Shift 4 positions to the right.

ESC, MM Abort the command.

any other key Don't shift.

AutoInd

Toggle automatic indentation mode on/off. Default is on.

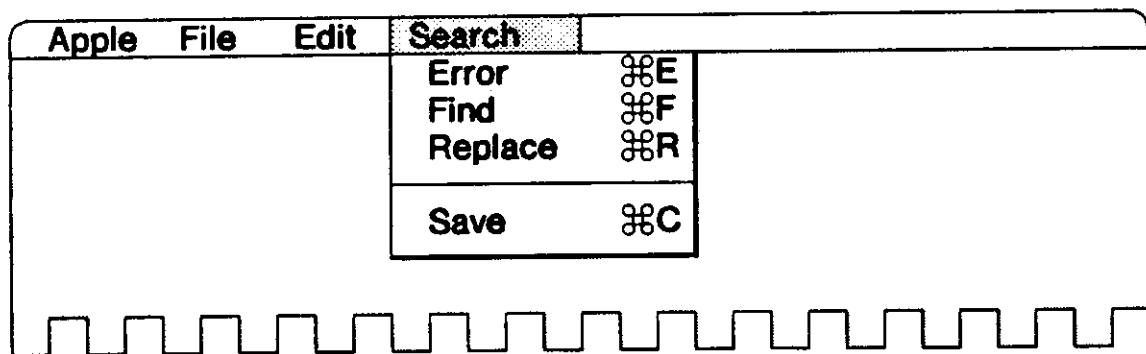
Search

Fig 3.6: Search Commands

Error

Search for the next (compiler detected) error starting at the caret position.

Find

Find a text portion starting at the caret position.

find>

text selected Find the selected text.

keyboard Enter text by keyboard.

EOL Terminate input.

DEL Delete last entered character.

	ESC, MR	Abort the command.
ML		Find next occurrence of a previously searched text.
ESC, MR		Abort the command.

Replace

Replace a text portion (to be searched for) by the internal buffer contents, starting at the caret position. The text to be found is specified as in the *Find* command. The new text has to be moved to the internal buffer (e.g. with *Save*). If the text is found the editor asks whether it should be replaced.

replace by buffer content?>

"y", "Y", ML	Replace and search again.
"n", "N", MR	Do not replace but search again.
"0", "1", ..., "9"	Replace 0, 1, ..., 9 times.
EOL	Replace all occurrences starting at the caret. Typing any key stops the replacing.
ESC	Abort the command.
SPC	Replace.
any other key	Do not replace.

Save

Copy a text portion to the internal buffer. If no text is selected the internal buffer is cleared (to allow to the replace command to delete a found string).

4. The Compiler

Niklaus Wirth 25-Oct-85, Werner Heiz, Hermann Seiler, 12-Aug-86

4.1. Introduction

This chapter describes the use of the Modula-2 compiler. For the language definition refer to the Modula-2 manual [1]. This compiler is a single-pass Modula-2 compiler generating native MC68000-code at a speed of about 1500 lines/minute on diskette or 3000 lines/minute on hard disk. Linking is not necessary as the code is completely relocatable and need not to be patched.

Glossary

compilation unit

Unit accepted by compiler for compilation, i.e. definition module or program module (see Modula-2 syntax).

definition module

Part of a separate module specifying the exported objects.

program module

Implementation part of a separate module (called implementation module) or main module.

source file

Input file of the compiler, i.e. a compilation unit. Default extension is "MOD".

symbol file

Compiler output file with symbol table information. This information is generated during compilation of a definition module. Assigned extension is "SBM".

reference file

Compiler output file with debugger information, generated during compilation of a program module. Assigned extension is "RFM".

object file

Compiler output file with the generated MC68000-code in MacMETH loader format. Assigned extension is "OBM".

4.2. Starting the Compiler

You start the compiler by selecting *Compile* from the MacMETH-menu:

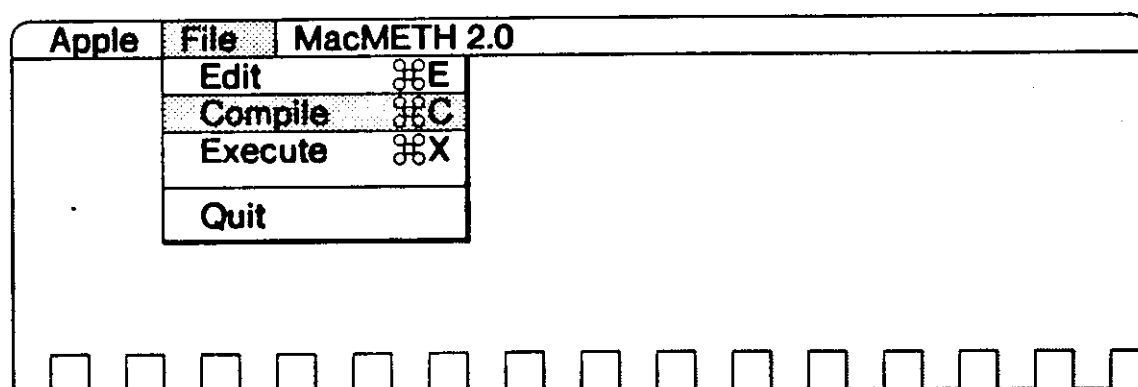


Fig. 4.1: Starting the Compiler

Now the Macintosh presents the following screen image:

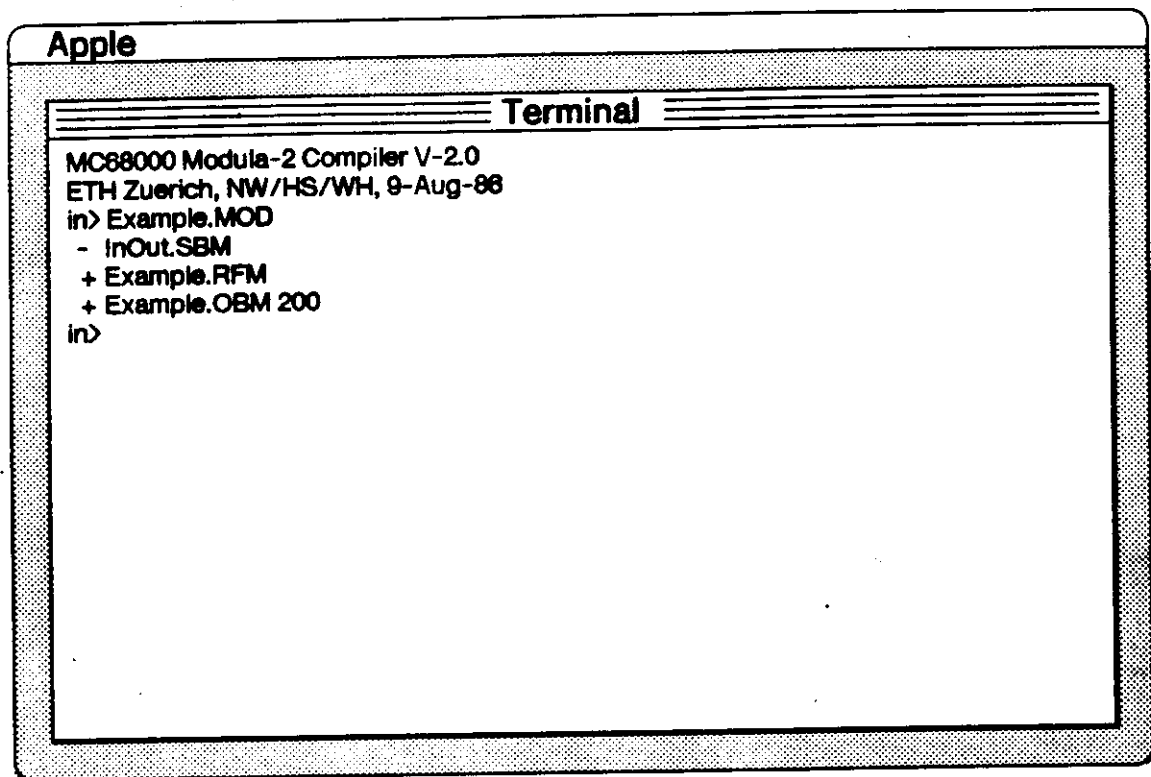


Fig. 4.2: Compiler Screen Image

You see the window "Terminal" on screen, possibly giving you a default file name for compilation (the last compiled file is the default one).

4.3. Compilation

Compilation of a Program Module

Press EOL to accept the name or enter an alternate name. If you end your file name with "." the extension "MOD" is appended. The compiler displays all files it consumes (i.e. imports) with the tag "-" and all files it produces with the tag "+". After successful compilation of a program module the number of generated code bytes is displayed. In case of an error the message "errors detected" appears instead. After compilation, the compiler again requests a file name for a further compilation. Terminate by typing ESC.

Compilation of a Definition Module

For definition modules the file name extension "DEF" is recommended. The definition part of a module must be compiled prior to its implementation part. Upon compilation of a definition module, a symbol file containing symbol table information is generated. This information is needed by the compiler in two cases:

- At compilation of the implementation part of the module.
- At compilation of another unit, importing objects from this separate module.

If a required symbol file is missing, the compilation process is stopped.

Compiler Output Files

Several files are generated by the compiler. Their file names are taken as the compilation unit's module name with the appropriate file name extension:

SBM	symbol file	err.LST	error file for user (text file)
RFM	reference file	err.DAT	error file for editor (data file)
OBM	object file		

The reference file is used by the symbolic debugger.

Module Key

With each compilation unit the compiler generates a so called module key. This key is unique and is needed to distinguish different compiled versions of the same module. The module key is written on the symbol file and on the object file.

For an implementation module the key of the associated definition module is adopted. The module keys of imported modules are also recorded on the generated symbol files and the object files.

Any mismatch of module keys belonging to the same module will cause an error message at compilation or loading time.

W A R N I N G

Recompilation of a definition module will produce a new symbol file with a new module key. In this case the implementation module and all units importing this module must be recompiled as well.

Recompilation of an implementation module does not affect the module key.

Compilation Options

The compiler optionally generates various redundancy checks. They can be enabled or disabled for each compilation by appending option characters to the source file name. The occurrence of an option character signals the inverse of its default value.

r	array index bound check and subrange check	default = on
v	arithmetic overflow check	default = off

Example: Name.MOD/v (all checks on); Name.MOD/vr (arithmetic checks on).

4.4. Program Execution

Programs are normally executed on top of the resident user environment MacMETH (although you can transform any Modula-2 program into a stand-alone Macintosh application). The command interpreter accepts a program name and causes the loader to load the module on the corresponding object file into memory and to start its execution.

If a program consists of several separate modules, no explicit linking is necessary. The object files generated by the compiler are merely ready to be loaded. Besides the main module, the module which is called to be executed and therefore constitutes the main program, all

modules which are directly or indirectly imported are loaded. The loader establishes the links between the modules.

Usually some of the imported modules are part of the already loaded, resident MacMETH system (e.g. module FileSystem). In this case the loader sets up the links to these modules, but prohibits their reinitialization. A module cannot be loaded twice.

After termination of the program, all separate modules which have been loaded together with the main module are removed from the memory. More details concerning program execution are given in chapter 7 (module *System*).

MacMETH also supports some kind of a program stack. A program may call another program, which will be executed on the top of the calling program. After termination of the called program, control will be returned to the calling program. For more details refer to chapter 7 (module *System*).

4.5. The Implemented Language

Standard Types

INTEGER

The value range of type INTEGER is [-32768..32767] (2 bytes). Sign inversion is an operation within constant expressions. Therefore the compiler does not allow the direct definition of -32768. This value must be computed indirectly, e.g. -32767-1.

CARDINAL

The value range of type CARDINAL is [0..65535] (2 bytes).

REAL

Values of type REAL are represented in 4 bytes. The value range expands from -3.38E38 to +3.38E38.

CHAR

The character set of type CHAR is defined according to the ISO-ASCII standard with ordinal values in the range [0..255] (1 byte). The compiler processes character constants in the range [0C..377C].

BITSET

The type BITSET is defined as SET OF [0..15] (2 bytes). Consider that sets are represented from the low order bits to the high order bits, i.e. {0} corresponds to the ordinal value 1.

LONGINT

The value range of type LONGINT is [-2147483648D..2147483647D] (4 bytes). Constants of type LONGINT and LONGCARD must have the suffix letter "D".

LONGCARD

The value range of type LONGCARD is [0D..4294967295D] (4 bytes).

LONGREAL

Values of type LONGREAL are represented in 8 bytes. The value range expands from -1.79E308 to +1.79E308.

Standard Procedures

Standard procedures are predefined (need not to be imported). Some are *generic* procedures that cannot be explicitly declared, i.e. they apply to classes of operand types or have several possible parameter list forms. Standard procedures are:

ABS(x)

Absolute value; result type = argument type.

CAP(ch)

If ch is a lower case letter, the corresponding capital letter;
otherwise the same letter.

CHR(x)

The character with ordinal number x.

FLOAT(x)

x of type INTEGER or LONGINT represented as a value of type REAL.

FLOATD(x)

x of type INTEGER or LONGINT represented as a value of type LONGREAL.

HIGH(a)

High index bound of array a (result type is INTEGER).

ODD(x)

$x \bmod 2 \neq 0$.

ORD(x)

Ordinal number (of type INTEGER) of x in the set of values defined by type T of x. T is any enumeration type, CHAR, INTEGER or CARDINAL.

TRUNC(x)

Real number x truncated to its integral part (of type INTEGER).

TRUNC D(x)

Longreal number x truncated to its integral part (of type LONGINT).

DEC(x); DEC(x, n)

$x := x - 1$; $x := x - n$. The type of x must be scalar.

EXCL(s, i)

$s := s - \{i\}$.

HALT

Terminate program execution (can be used as programmed break-point in MacMETH).

INC(x); INC(x, n)

$x := x + 1$; $x := x + n$. The type of x must be scalar.

INCL(s, i)

$s := s + \{i\}$.

MAX(T)

T is any scalar type (including REAL). Result is the type's maximum value.

MIN(T)

T is any scalar type (including REAL). Result is the type's minimum value.

SIZE(x); SIZE(T)

Number of bytes (INTEGER) the variable or type occupies in memory.

The Module SYSTEM

Explicitly system-dependent features are imported from module SYSTEM. Although it lies in the nature of this module that there cannot be a standard for its contents, it typically exports the type ADDRESS which is compatible with all pointer types. Here the type ADDRESS is also compatible with LONGCARD for address arithmetic (instead of with CARDINAL). The new type BYTE represents the unit of addressable storage.

```
DEFINITION MODULE SYSTEM; (* for MC68000 *)
  TYPE ADDRESS = POINTER TO BYTE;
  (* compatible with type LONGCARD and with all pointer types *)
  BYTE;      (* smallest addressable unit, uninterpreted, TSIZE(BYTE) = 1,
              ARRAY OF BYTE is compatible with everything *)
  WORD;      (* two uninterpreted consecutive bytes beginning at an even
              address, compatible with all types having size 2,
              (the MC68000 generates traps on odd word addresses) *)

  (* T subsequently denotes any simple type of size <= 4 bytes *)
  PROCEDURE ADR(VAR x: AnyType): ADDRESS;      (* address of variable x *)
  PROCEDURE INLINE(x: WORD);                  (* put x in instruction stream *)
  PROCEDURE REG(reg: INTEGER): LONGINT;        (* return MC68000 reg contents,
                                              reg must be a constant,
                                              reg: 0-7=D0-D7, 8-15=A0-A7 *)

  PROCEDURE SETREG(reg: INTEGER; val: T);      (* set register to value *)
  PROCEDURE TSIZE(AnyType): INTEGER;          (* return number of bytes *)
  PROCEDURE LONG(hi, lo: CARDINAL): LONGINT;   (* construct LONGINT *)
  PROCEDURE LONG(x: T): LONGINT;              (* second form of LONG function *)
  PROCEDURE SHORT(x: LONGINT): INTEGER;        (* return lower word of LONGINT *)
  PROCEDURE SHIFT(x: T; n: INTEGER): T;        (* x shifted by n bits, n>0 left *)
  PROCEDURE VAL(AnyType; x: AnyType): AnyType; (* type conversion *)
END SYSTEM.
```

The generic function procedure VAL(T, x) is effectively a replacement for type transfer functions T(x). Its value is x, interpreted as type T. No code is generated for this pseudo-procedure. Its explicit import is to make the uses of machine-dependent type transfers explicit and more readily locatable.

Differences and Restrictions

For the implementation of Modula-2 on the Macintosh some differences and restrictions must be considered.

Type Conversions

Genuine difficulties in porting a Modula program from one computer to another appear where machine-dependent features of Modula were used. They are supposedly highlighted by imports from module SYSTEM, an import that everybody knows should only be taken as a last resort. Another, much less obvious and therefore easily abused machine-dependence is the type transfer function. The ease with which it allows the (perhaps inadvertant) hiding of machine-dependence makes it a highly controversial feature, and this compiler does therefore not support it. Type transfers can, however, be obtained through the use of VAL (imported from SYSTEM).

Assignment Compatibility

The following types are assignment compatible to each other (overflow is checked):

(INTEGER, CARDINAL, LONGINT, LONGCARD), and (REAL, LONGREAL).

Procedures

The compiler also offers a code procedure declaration. It can be used in definition and implementation modules and serves to introduce procedures implemented by supervisor calls. The code number *n* specifies a MC68000 opcode (typically a trap instruction) inserted as a word in the instruction stream where called. Evidently, such definitions are provided together with the operating system used. The declaration format is

PROCEDURE P(parameter list); CODE *n*;

Procedures referenced before declaration must be declared prior by a FORWARD declaration. The format is

PROCEDURE P(parameter list); FORWARD;

The corresponding procedure must have the full header repeated and must lie at the same nesting level as the FORWARD declaration.

Forward References

No forward references are permitted, except in definitions of pointer types and in forward procedure declarations.

Function procedures

The result type size of a function procedure must be 1, 2, 4, or 8 (that includes all simple types).

Data Size

The maximum total global (static) data size must be less than 32 kBytes per compilation unit. No limit on total data size over all modules or dynamic data size.

Code Size

The maximum code size must be less than 32 kBytes per compilation unit. The total code size over all modules has no limit (e.g. the compiler has 86 kBytes).

Index types in array declarations

The index type must be a subrange type.

Standard functions, procedures, types, and absolute variables

The procedures NEW, DISPOSE, TRANSFER, NEWPROCESS, and the type PROCESS are not predeclared (although defined as standard objects in earlier reports on the language). They will be implemented in separate modules.

The function SIZE is a standard procedure (globally defined), and it is identical to the function TSIZE defined in module SYSTEM. Its argument is a type or a variable; the result is the size of the type (of the variable) in number of bytes required for the variable or for a variable of that type.

IMPORT System; VAR p: POINTER TO T;

NEW(p) must be written as System.Allocate(p, SIZE(T))

Absolute variables are not supported (access them via pointers).

Subranges

The bounds of a subrange must be less than 2^{15} in absolute value and the difference $\text{MAX}(\text{Subrange}) - \text{MIN}(\text{Subrange})$ must be less than 2^{15} .

Opaque types

If a type *T* is declared in a definition module to be opaque, it cannot (in the corresponding implementation module) be declared as equal to another, named type.

Enumeration Types

An enumeration may have at most 256 constants.

Sets

A set may have at most 16 elements.

Procedures declared in definition modules

If a procedure (heading) is declared in a definition module, its body must be declared in the corresponding implementation module proper; it cannot be declared in an inner, local module.

Some Programming Hints and Implementation Notes

1. The MC68000 uses byte addressing. However, data are transferred to and from memory in 16-bit words. Each type has an alignment factor k . Variables are aligned by the compiler to lie at an address a , such that $a \text{ MOD } k = 0$ (otherwise the MC68000 would generate an odd address trap). Since allocation is sequential, i.e. variables are allocated in the order of their textual occurrence, the least amount of storage gets wasted through alignment, if declarations are grouped according to size. The same holds for record fields, and in this case is even more important. The following are the sizes and alignment factors of types:

Type	Size	Alignment Factor
CHAR, BOOLEAN, enumerations, BYTE	1	1
INTEGER, CARDINAL, WORD, BITSET, sets	2	2
LONGINT, REAL, pointers, procedures	4	2
LONGREAL	8	2
arrays, records	multiple of 2	2

2. The statements INC(n), DEC(n), INCL(s,n), EXCL(s,n) generate considerably denser code than their equivalents $n := n + 1$, $n := n - 1$, $s := s + \{n\}$, $s := s - \{n\}$, even in the case that n or s are simple variables. INC and DEC accept one or two parameters.

3. Access to so-called intermediate-level variables is slower and requires more code than access to local or global variables. Such accesses should therefore be made only after careful justification. Intermediate-level variables should be considered as implicit, additional procedure parameters.

4. The MC68000 doesn't supply all necessary instructions needed to implement Modula-2. A runtime support module is therefore necessary. The module "System" (see 7.13.) provides the missing operations. It is a normal Modula-2 module consisting of a definition and an implementation part (that can be and is translated by the compiler). If the compiler has to generate code for an operation the MC68000 doesn't support (i.e. real arithmetic) it actually generates an external call to the module "System" which always is imported implicitly. In fact, instead of writing $x := y + z$ (x, y, z : REAL) you could write $x := \text{System.FADDS}(y, z)$ as well.

4.6. Modula-2 Syntax

```

ident = letter {letter | digit}.
number = integer | real.
integer = digit {digit} ["D"] | octalDigit {octalDigit} ("B"|"C") |
  digit {hexDigit} "H".
real = digit {digit} "." {digit} [ScaleFactor].
ScaleFactor = "E" ["+"|"-" ] digit {digit}.
hexDigit = digit ["A"|"B"|"C"|"D"|"E"|"F"].
digit = octalDigit | "8"|"9".
octalDigit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7".
string = "" {character} "" | "" {character} "".
qualident = ident { "." ident }.
ConstantDeclaration = ident "=" ConstExpression.
ConstExpression = expression.
TypeDeclaration = ident "=" type.
type = SimpleType | ArrayType | RecordType | SetType |
  PointerType | ProcedureType.
SimpleType = qualident | enumeration | SubrangeType.
enumeration = "(" IdentList ")".
IdentList = ident { "," ident }.
SubrangeType = [qualident] "[" ConstExpression ".." ConstExpression "]".
ArrayType = ARRAY SimpleType { "," SimpleType } OF type.
RecordType = RECORD FieldListSequence END.
FieldListSequence = FieldList { ";" FieldList }.
FieldList = [IdentList ":" type |
  CASE [ident] ":" qualident OF variant { "[" variant }
  [ELSE FieldListSequence] END].
variant = [CaseLabelList ":" FieldListSequence].
CaseLabelList = CaseLabels { "," CaseLabels }.
CaseLabels = ConstExpression [".." ConstExpression].
SetType = SET OF SimpleType.
PointerType = POINTER TO type.
ProcedureType = PROCEDURE [FormalTypeList].
FormalTypeList = "(" [[VAR] FormalType { "," [VAR] FormalType } "]" [ ":" qualident ].
VariableDeclaration = IdentList ":" type.
designator = qualident { "." ident | "(" ExpList ")" | "↑" }.
ExpList = expression { "," expression }.
expression = SimpleExpression [relation SimpleExpression].
relation = "=" | "<" | "<=" | ">" | ">=" | IN .
SimpleExpression = ["+"|"-" ] term { AddOperator term }.
AddOperator = "+" | "-" | OR .
term = factor { MulOperator factor }.
MulOperator = "*" | "/" | DIV | REM | MOD | AND | "&" .
factor = number | string | set | designator [ActualParameters] |
  "(" expression ")" | NOT factor | "~" factor.
set = [qualident] "{" [element { "," element } } ".
element = ConstExpression [".." ConstExpression].
ActualParameters = "(" [ExpList] ")".
statement = [assignment | ProcedureCall |
  IfStatement | CaseStatement | WhileStatement |

```

RepeatStatement | LoopStatement | ForStatement |
 WithStatement | EXIT | RETURN [expression]].
 assignment = designator ":" = " expression.
 ProcedureCall = designator [ActualParameters].
 StatementSequence = statement {";" statement}.
 IfStatement = IF expression THEN StatementSequence
 {ELSIF expression THEN StatementSequence}
 [ELSE StatementSequence] END.
 CaseStatement = CASE expression OF case {"|" case}
 [ELSE StatementSequence] END.
 case = [CaseLabelList ":" StatementSequence].
 WhileStatement = WHILE expression DO StatementSequence END.
 RepeatStatement = REPEAT StatementSequence UNTIL expression.
 ForStatement = FOR ident ":" = " expression TO expression
 [BY ConstExpression] DO StatementSequence END.
 LoopStatement = LOOP StatementSequence END.
 WithStatement = WITH designator DO StatementSequence END .
 ProcedureDeclaration = ProcedureHeading ";" (block ident | FORWARD).
 ProcedureHeading = PROCEDURE ident [FormalParameters].
 block = {declaration} [BEGIN StatementSequence] END.
 declaration = CONST {ConstantDeclaration ":"} |
 TYPE {TypeDeclaration ":"} |
 VAR {VariableDeclaration ":"} |
 ProcedureDeclaration ";" | ModuleDeclaration ":".
 FormalParameters =
 "(" [FPSection {";" FPSection}] ")" [";" qualident].
 FPSection = [VAR] IdentList ":" FormalType.
 FormalType = [ARRAY OF] qualident.
 ModuleDeclaration =
 MODULE ident [priority] ":" {import} [export] block ident.
 priority = "[" ConstExpression "]".
 export = EXPORT [QUALIFIED] IdentList ":".
 import = [FROM ident] IMPORT IdentList ":".
 DefinitionModule = DEFINITION MODULE ident ":"
 {import} {definition} END ident ".".
 definition = CONST {ConstantDeclaration ":"} |
 TYPE {ident ["=" type] ":"} |
 VAR {VariableDeclaration ":"} |
 ProcedureHeading ":".
 ProgramModule = MODULE ident [priority] ":" {import} block ident "." .
 CompilationUnit = DefinitionModule | [IMPLEMENTATION] ProgramModule.

4.7. Compiler Error Messages

Syntax errors

- 10 identifier expected
- 11 , comma expected
- 12 ; semicolon expected
- 13 : colon expected
- 14 . period expected
- 15) right parenthesis expected
- 16] right bracket expected
- 17 } right brace expected
- 18 = equal sign expected
- 19 := assignment expected
- 20 END expected
- 21 .. ellipsis expected
- 22 (left parenthesis expected
- 23 OF expected
- 24 TO expected
- 25 DO expected
- 26 UNTIL expected
- 27 THEN expected
- 28 MODULE expected
- 29 illegal digit, or number too large
- 30 IMPORT expected
- 31 factor starts with illegal symbol
- 32 identifier, (, or [expected
- 33 identifier, ARRAY, RECORD, SET, POINTER, PROCEDURE, (, or [expected
- 34 Type followed by illegal symbol
- 35 statement starts with illegal symbol
- 36 declaration followed by illegal symbol
- 37 statement part is not allowed in definition module
- 38 export list not allowed in program module
- 39 EXIT not inside a LOOP statement
- 40 illegal character in number
- 41 number too large
- 42 comment without closing *)
- 43
- 44 expression must contain constant operands only
- 45 control character within string

Undefined

- 50 identifier not declared or not visible

Class and type errors

- 51 object should be a constant
- 52 object should be a type
- 53 object should be a variable
- 54 object should be a procedure
- 55 object should be a module

- 56 type should be a subrange
- 57 type should be a record
- 58 type should be an array
- 59 type should be a set
- 60 illegal base type of set
- 61 incompatible type of label or of subrange bound
- 62 multiply defined case (label)
- 63 low bound > high bound
- 64 more actual than formal parameters
- 65 fewer actual than formal parameters

- 66 - 73 mismatch between parameter list D in definition and I in implementation module
- 66 more parameters in I than in D
- 67 parameters with equal types in I have different types in D
- 68 mismatch between VAR specifications
- 69 mismatch between type specifications
- 70 more parameters in D than in I
- 71 mismatch between result type specifications
- 72 function in D, pure procedure in I
- 73 procedure in D has parameters, but not in I

- 74 code procedure cannot be declared in definition module
- 75 illegal type of control variable in FOR statement
- 76 procedure call of a function
- 77 identifiers in heading and at end do not match
- 78 redefinition of a type that is declared in definition part
- 79 imported module not found
- 80 unsatisfied export list entry
- 81 illegal type of procedure result
- 82 illegal base type of subrange
- 83 illegal type of case expression
- 84 writing of symbol file failed
- 85 keys of imported symbol files do not match
- 86 error in format of symbol file
- 88 symbol file not successfully opened
- 89 procedure declared in definition module, but not in implementation

Implementation restrictions of compiler

- 90 in {a..b}, if a is a constant, b must also be a constant
- 91 identifier buffer overflow
- 92 too many cases
- 93 too many exit statements
- 94 index type of array must be a subrange
- 95 subrange bound must be less than 2^{15}
- 96 too many global modules
- 97 too many procedure in definition module
- 98 too many structure elements in definition module
- 99 too many variables, or record too large

Multiple definition

100 multiple definition within the same scope

Class and type incompatibilities

- 101 illegal use of type
- 102 illegal use of procedure
- 103 illegal use of constant
- 104 illegal use of type
- 105 illegal use of procedure
- 106 illegal use of expression
- 107 illegal use of module
- 108 constant index out of range
- 109 indexed variable is not an array, or the index has the wrong type
- 110 record selector is not a field identifier
- 111 dereferenced variable is not a pointer
- 112 operand type incompatible with sign inversion
- 113 operand type incompatible with NOT, AND, OR
- 114 $x \text{ IN } y$: $\text{type}(x) \neq \text{basetype}(y)$
- 115 type of x cannot be the basetype of a set, or y is not a set
- 116 $\{a..b\}$: type of either a or b is not equal to the base type of the set
- 117 incompatible operand types
- 118 operand type incompatible with $*$
- 119 operand type incompatible with $/$
- 120 operand type incompatible with DIV
- 121 operand type incompatible with MOD
- 122 operand type incompatible with AND
- 123 operand type incompatible with $+$
- 124 operand type incompatible with $-$
- 125 operand type incompatible with OR
- 126 operand type incompatible with relation

- 127-131: assignment of a procedure P to a variable of type T
- 127 procedure must have level 0
- 128 result type of P does not match that of T
- 129 mismatch of a parameter of P with the formal type list of T
- 130 procedure has fewer parameters than the formal type list
- 131 procedure has more parameters than the formal type list

- 132 assignment of a negative integer to a cardinal variable
- 133 incompatible assignment
- 134 assignment to non-variable
- 135 type of expression in IF, WHILE, UNTIL clause must be BOOLEAN
- 136 call of an object which is not a procedure
- 137 type of VAR parameter is not identical to that of actual parameter
- 138 value assigned to subrange variable is out of bounds
- 139 type of RETURN expression differs from procedure type
- 140 illegal type of CASE expression
- 141 step in FOR clause cannot be 0
- 142 illegal type of control variable

- 143
- 144 incorrect type of parameter of standard procedure
- 145 this parameter should be a type identifier
- 146 string is too long
- 147 incorrect priority specification
- 148
- 149

Name collision

- 150 exported identifier collides with declared identifier

Implementation restrictions of system

- 200 (not yet implemented)
- 201 integer too small for sign inversion
- 202 set element outside word range
- 203 overflow in multiplication
- 204 overflow in division
- 205 division by zero, or modulus with negative value
- 206 overflow in addition
- 207 overflow in subtraction
- 208 cardinal value assigned to integer variable too large
- 209 set size too large
- 210 array size too large
- 211 address too large (compiler error?)
- 212 character array component cannot correspond to VAR parameter
- 213 illegal store operation (compiler error?)
- 214 set range bounds must be constants
- 215 expression too complex (register overflow)

- 222 output file not opened (directory full?)
- 223 output incomplete (disk full?)
- 224 too many external references
- 225 too many strings
- 226 program too long

- 230 expression not loadable (implementation restriction)
- 231 expression not addressable (implementation restriction)
- 232 expression not allowed (implementation restriction)
- 233 illegal expression (implementation restriction)
- 234 register reservation error
- 235 illegal selector for constant index / field
- 236 too many WITH nested (> 4)
- 237 illegal operand (implementation restriction)
- 238 illegal size of operand (implementation restriction)
- 239 type should be LONGREAL
- 240 parameter should be dynamic array parameter
- 241 illegal type for floating point operation
- 244 implementation restriction for floating point comparison
- 300 - 399 (implementation restriction, compiler error?)

5. The Debugger

Werner Heiz, 12- Aug-86

5.1. Introduction

The debugger is a tool having the task to assist the programmer in finding the reasons for execution errors in his program. The programmer is served best if he is allowed to debug on the same abstraction level in which he originally created his program. That means he should be able to view variables by name accompanied by types and interpreted values, he shouldn't have to struggle with addresses and their binary contents as it is the case in many debuggers in current use. The debugger's main task and its relationship to the compiler is illustrated in the following diagram (Fig. 5.1):

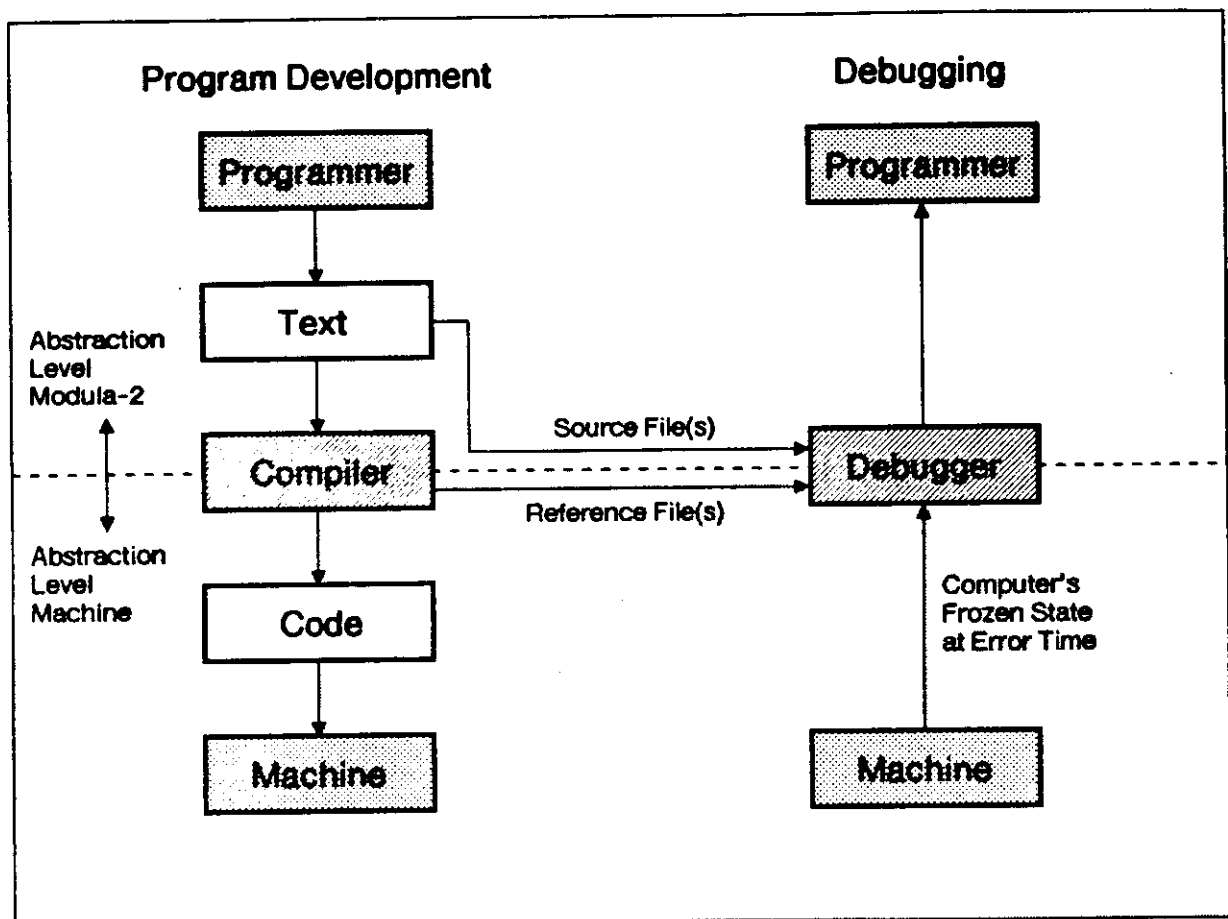


Fig. 5.1: Debugger Environment

On the left side the program development process is shown. The programmer writes his program in a high-level language, the compiler translates this text into machine code which, eventually, is loaded into the machine and executed. Thus, the compiler performs a mapping from a representative of the Modula-2 world into one of the machine world leaving the semantics invariant.

On the right side the program debugging process is shown. Suppose the program execution leads to an error. The programmer wants to know the error position in the program text and find the logical error (reason) causing the program running into this error (symptom). The debugger maps the machine state in binary form into a form being easily comprehensible to the programmer. Because during the mapping that the compiler performs information is lost

the debugger needs additional information: program text and compilation information.

The compiler generates not only a code file but also a reference file containing information like variable names, types, declaration levels, addresses and such. The debugger also needs the source file to show the programmer the error position in the program text. It is important to recognize that the debugger does *not* have to look at the machine code, it is static and known anyway. The debugger is only interested in the data, they are dynamic and only known at program execution time. The debugger is in contrast to the compiler not a translator but an interpreter (of the machine's state in binary form) as the mapping process is continuous and interactive.

5.2. Starting the Debugger

If your program runs into an execution error an alert box is displayed by the Macintosh as follows:

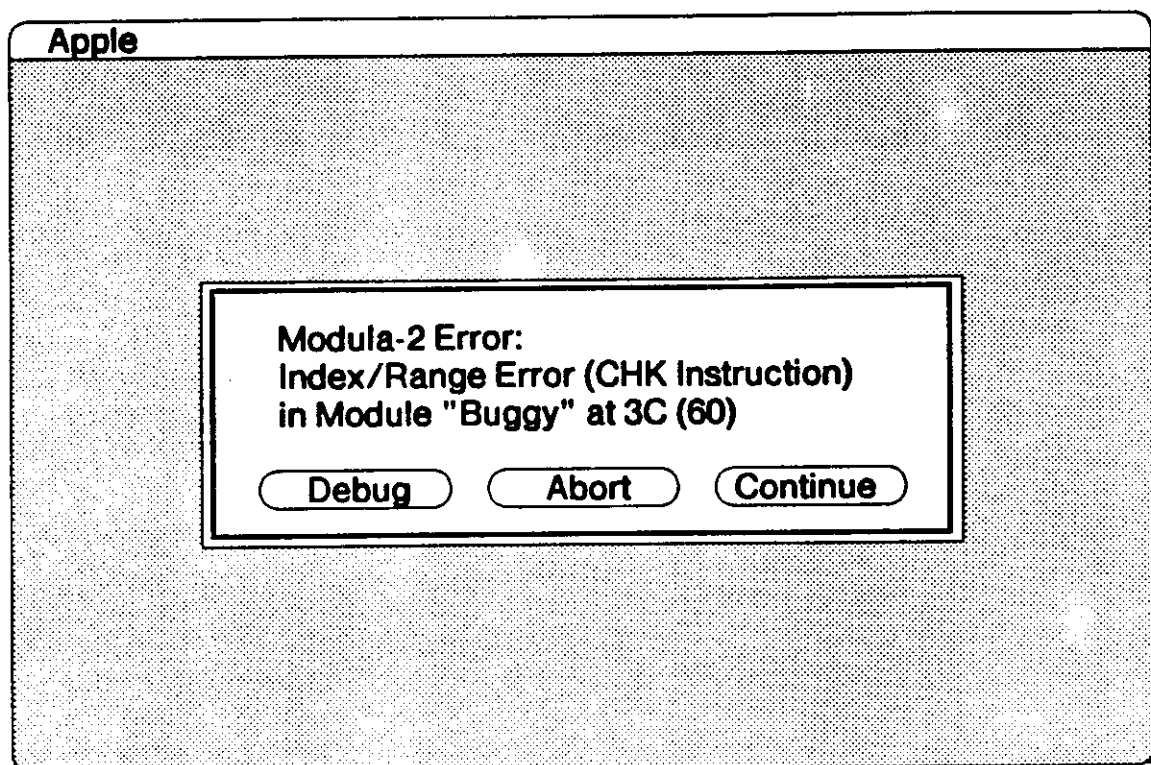


Fig. 5.2: Runtime Error

The box gives error type, the module the error occurred in, and the relative address in that module in form of a hexadecimal and decimal number. You have three options: enter the debugger, abort the program or continue execution (not all error types allow you to continue). Selecting *Debug* starts the debugger which presents the following screen image:

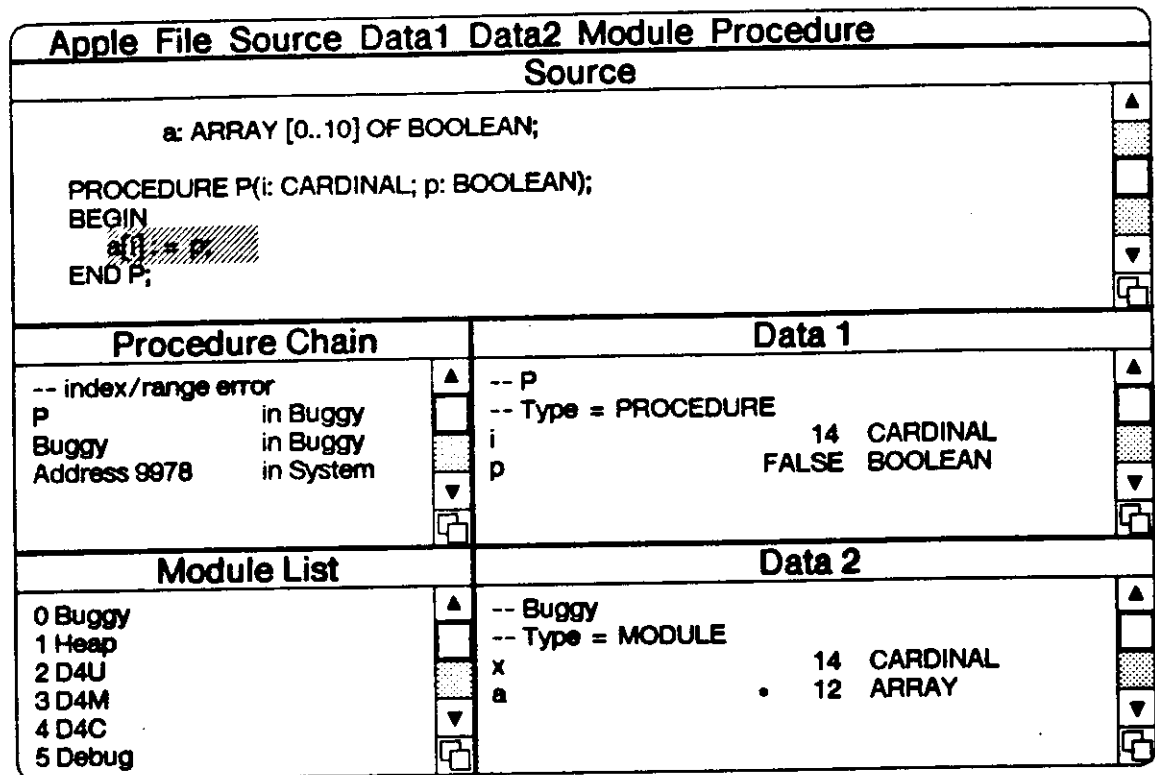


Fig. 5.3: Debugger Screen Image

5 Windows become visible:

- **Source Window:** Contains the source listing with the erroneous statement high-lighted.
- **Procedure Chain Window:** Indicates the reason for the program crash and shows the calling sequence of the active procedures.
- **Data 1 Window:** Contains all the variables of the last called procedure.
- **Module List Window:** Lists all loaded modules.
- **Data 2 Window:** Contains all the variables of the module containing the offending procedure.

Subsequently we shall take a closer look at each window.

5.3. Global Commands

Each Window has a scroll bar on the right. Clicking inside the scroll bar causes the contents to scroll as in the editor. Selecting *Quit* from the *File* menu leaves the debugger.

The mouse buttons (ML and MR) are used as follows:

- ML: Display data of selected object (selecting = pointing and clicking).
- MR: Display source of selected object (if available).

5.4. Local Commands

The Source Window

The source window contains a listing which is selected through the procedure chain or the module list window. If selected through the procedure chain window the error position is highlighted.

ML: -

MR: -

Source Menu:	Ask On	Ask user for alternate file names if source file cannot be found. Normally the source file name is derived from the module name by adding ".MOD".
	Ask Off	Don't ask user for alternate file names.

The Data Windows

The two data windows display the data related with selected modules, procedures, arrays, records and pointers.

ML: Show further data of selected object if there is such.

Objects with further data are marked with "*".

The top line shows a path through which the present structure was reached.

The path can be retraced in reverse order by selecting a previous component.

MR: -

Data Menus:	Expand	Expand window.
	Shrink	Shrink window.

The Module List Window

The module list window shows the list of loaded modules.

ML: Show global data of selected module.

MR: Show source listing of selected module.

Module Menu:	Data 1	Use data window 1 to display global data of modules.
	Data 2	Use data window 2 to display global data of modules.
	DEF	Show the definition listing of future selected modules.
	MOD	Show the implementation listing of future selected modules.

The Procedure Chain Window

The procedure chain window shows the error diagnostic and the calling sequence of all active procedures.

ML: Show local data of selected procedure.

MR: Show source listing of selected procedure.

Procedure Menu:	Data 1	Use data window 1 to display local data of procedures.
	Data 2	Use data window 2 to display local data of procedures.

6. Utility Programs

· Christian Vetterli, 26-Feb-86, Werner Heiz, 7-Apr-86

6.1. The Linker

The program *link* collects the codes of separate modules of a program and writes them on one file. The program *link* is called *linker* in this chapter.

Upon compilation of a separate module, the code generated by the Modula-2 compiler is written on an *object file*. An object file may be loaded by the loader of MacMETH directly.

As a program usually consists of several separate modules, the loader has to read the code of the modules from several object files which are searched according to a *default strategy*. On one hand, this is time consuming because several files must be searched, on the other hand, it could be useful to substitute a module from a file with a non-default name. These are some reasons for having a linker program.

The linker simulates the loading process and collects the codes of all modules which are, directly or indirectly, imported by the so-called *main module*, i.e. the module which constitutes the main program. The linker applies the same default strategy as the loader to find an object file. A file name is derived from (the first 16 characters of) the module name.

Example: module name: **AnyModule**, default file name: **AnyModule.OBM**.

The linker first prompts for the object file of the main module (default extension "OBM"). Next, it displays the name of the main module. If the file already contains some linked modules, the names of these modules are displayed next. Afterwards, a name of a not yet linked imported module is displayed, followed by the file name of the corresponding object file. On the next lines the names of the modules linked from this file are listed. This is repeated until all imported modules are linked.

```

Linker VC 1.0 26.2.86 for MacMETH
object file> delete.OBM
  Delete                                main module
NameSearch: NameSearch.OBM            default file name
  NameSearch
Options: Options.OBM                   default file name
  Options
  FileNames                            module was linked to Options
end of linkage

```

After successful linking, all linked modules are written on the object file of the main module.

The linker accepts three program options when it prompts for the main module:

/Q (query)

If this option is set, the linker also prompts for the file names of the imported modules. Type a file name (default extension "OBM") or simply press the RETURN key to apply the default strategy. A prompt is repeated until an adequate object file is found, or the ESC key is pressed. The latter means that this module should not be linked.

With the query option the linker also asks whether or not a module on a object file should be linked. Type "y" to accept the module, otherwise type "n".

```

object file> delete.OBM/q             query option set
Delete

```

```

NameSearch> NameSearch.new.OBM  own file substituted
      NameSearch  ?  yes
Options> Options.OBM           default file name
      Options    ?  yes
      FileNames  ?  no           module not linked from this file
FileNames> FileNames.own.OBM
      FileNames  ?  yes

```

/A (application)

If this option is set, a standalone application can be generated (activated by a double click on the Macintosh). The module System and the resources are copied from the linker itself (this way you could transform the compiler into a standalone application!).

```

object file> delete.OBM/a      application option set
      Delete
NameSearch: NameSearch.OBM     default file name
      NameSearch
Options: Options.OBM           default file name
      Options
      FileNames                module was linked to Options
end of linkage

```

/S (application from System, used for bootstrapping)

If this option is set, the linker prompts for the file names of the imported modules as it does with option Q, but it also prompts for the module System. No resources are included in the generated application!

```

object file> delete.OBM/s      system option set
      Delete
NameSearch> NameSearch.new.OBM own file substituted
      NameSearch  ?  yes
Options> Options.OBM           default file name
      Options    ?  yes
      FileNames  ?  no           module not linked from this file
FileNames> FileNames.own.OBM
      FileNames  ?  yes
System> System.OBM
      System     ?  yes           application is generated using module System
end of linkage

```

6.2. The Decoder

The program *dec68k* disassembles an object file.

The program reads an object code file and generates a textfile with mnemonics for the machine instructions. It respects the structure of the object file as generated by the compiler.

The program prompts for the name of the input file. Default extension is "OBM".

```
in> Buggy.OBM
out>Buggy.DEC
```

The intended usage of this program is to check the compiler after modifications of the code generation; however this program may be used also to learn about code generation. In production there is no need to know the code generated by the compiler.

Example:

Input: Object file of the module "Buggy"

```
MODULE Buggy: (* W. Heiz, 20-Nov-85 *)

  VAR  x: CARDINAL;
        a: ARRAY [0..10] OF BOOLEAN;

  PROCEDURE P(i: CARDINAL; p: BOOLEAN);
  BEGIN
    a[i] := p;
  END P;

BEGIN (* Buggy *)
  x := 12;
  P(0, TRUE);
  P(x, FALSE);
END Buggy.
```

Output: Decoded object file

MC68000 Decoder V-1.1, WH 14.1.86

```
HEADER
  Version   =      0
  Key, Name = AC62 01BA 7378, Buggy
  CodeSize  =     112
  DataSize  =      42
  ConstSize =      0
  Procs     =      1
  Mods      =      2
```

```
IMPORT
  Key, Name = 0000 0000 0000, System
```

```
CODE
  0: 4E71  NOP
```

```

2: 4E71  NOP
4: 2F0C  MOVE.L  A4, -(A7)
6: 287A  MOVE.L  -8(PC), A4
10: 4E56  LINK      A6, #0
14: 08EC  BSET      #0, -2(A4)
20: 6706  BEQ       6  abs=28
22: 4E5E  UNLK      A6
24: 285F  MOVE.L  (A7)+, A4
26: 4E75  RTS
28: 266C  MOVE.L  8(A4), A3
32: 266B  MOVE.L  0(A3), A3
36: 4E93  JSR      (A3)
38: 6000  BRA      40  abs=80
42: 2F0C  MOVE.L  A4, -(A7)
44: 287A  MOVE.L  -46(PC), A4
48: 4E56  LINK      A6, #0
52: 342E  MOVE.W  14(A6), D2
56: 45BC  CHK      #10, D2
60: 47EC  LEA      -42(A4), A3
64: 17AE  MOVE.B  12(A6), 0(A3, D2.W)
70: 4E5E  UNLK      A6
72: 285F  MOVE.L  (A7)+, A4
74: 265F  MOVE.L  (A7)+, A3
76: 588F  ADDQ.L  #4, A7
78: 4ED3  JMP      (A3)
80: 397C  MOVE.W  #12, -30(A4)
86: 4267  CLR.W   -(A7)
88: 1F3C  MOVE.B  #1, -(A7)
92: 6100  BSR     -52  abs=42
96: 3F2C  MOVE.W  -30(A4), -(A7)
100: 4227  CLR.B   -(A7)
102: 6100  BSR     -62  abs=42
106: 4E5E  UNLK      A6
108: 285F  MOVE.L  (A7)+, A4
110: 4E75  RTS

```

DATA

Entry Table:

4

Strings:

no strings

7. Library Modules

7.1. Introduction

This chapter describes a collection of some commonly used library modules on the Macintosh. For each library module a symbol file and an object file is stored on the disk. The file names are derived from (the first 16 characters of) the module name ending with the extension "SBM" for symbol files and the extension "OBM" for object files. It is possible that some object files are pre-linked and therefore also contain the code of the imported modules.

Example:

Module name	FileNames
Symbol file name	FileNames.SBM
Object file name	FileNames.OBM

List of the Library Modules

InOut	Simple handling of formatted input/output	7.2.
RealInOut, LongRealInOut	Formatted input/output of real numbers	7.3.
MathLib, LongMathLib	Basic mathematical functions	7.4.
FileSystem	Basic file input/output procedures	7.5.
Terminal, TerminalIn, TerminalOut	Basic input/output procedures	7.6.
Printer	Basic printer procedures	7.7.
CursorMouse	Mouse handling and cursor tracking	7.8.
Menu	Command selection from a menu	7.9.
Windows	Window handling (contents irrelevant)	7.10.
TextWindows	Text window handling	7.11.
GraphicWindows	Graphic window handling	7.12.
EventBase	Simple event scheduler	7.13.
System	Heart of MacMETH (loader, mem. manag., etc.)	7.14.

7.2. InOut

Niklaus Wirth, 15-May-82, rev. 20-Jun-82

Library module for formatted input/output on terminal or files. A description of this module is included to the Modula-2 manual [1].

Definition Module

```

DEFINITION MODULE InOut;      (*NW 20.6.82*)
  FROM SYSTEM IMPORT WORD;
  FROM FileSystem IMPORT File;

  CONST EOL = 15C;
  VAR Done: BOOLEAN;
      termCH: CHAR; (*terminating character in ReadInt, ReadCard*)
      in, out: File; (*for exceptional cases only*)

  PROCEDURE OpenInput(defext: ARRAY OF CHAR);
    (*request a file name and open input file "in".
     Done := "file was successfully opened".
     If open, subsequent input is read from this file.
     If name ends with ".", append extension defext*)

  PROCEDURE OpenOutput(defext: ARRAY OF CHAR);
    (*request a file name and open output file "out"
     Done := "file was successfully opened".
     If open, subsequent output is written on this file*)

  PROCEDURE CloseInput;
    (*closes input file; returns input to terminal*)

  PROCEDURE CloseOutput;
    (*closes output file; returns output to terminal*)

  PROCEDURE Read(VAR ch: CHAR);
    (*Done := NOT in.eof*)

  PROCEDURE ReadString(VAR s: ARRAY OF CHAR);
    (*read string, i.e. sequence of characters not containing
     blanks nor control characters; leading blanks are ignored.
     Input is terminated by any character <= " ";
     this character is assigned to termCH.
     DEL is used for backspacing when input from terminal*)

  PROCEDURE ReadInt(VAR x: INTEGER);
    (*read string and convert to integer. Syntax:
     integer = ["+"|"-"] digit {digit}.
     Leading blanks are ignored.
     Done := "integer was read"*)

  PROCEDURE ReadCard(VAR x: CARDINAL);
    (*read string and convert to cardinal. Syntax:
     cardinal = digit {digit}.
     Leading blanks are ignored.
     Done := "cardinal was read"*)

```

```

PROCEDURE ReadWrd(VAR w: WORD);
  (*Done := NOT in.eof*)

PROCEDURE Write(ch: CHAR);

PROCEDURE WriteLn;    (*terminate line*)

PROCEDURE WriteString(s: ARRAY OF CHAR);

PROCEDURE WriteInt(x: INTEGER; n: CARDINAL);
  (*write integer x with (at least) n characters on file "out".
    If n is greater than the number of digits needed,
    blanks are added preceding the number*)

PROCEDURE WriteCard(x,n: CARDINAL);
PROCEDURE WriteOct(x,n: CARDINAL);
PROCEDURE WriteHex(x,n: CARDINAL);
PROCEDURE WriteWrd(w: WORD);
END InOut.

```

Imported Library Modules

```

Terminal
FileSystem

```

7.3. RealInOut, LongRealInOut

Beat Stamm, 26-May-86

Library module for formatted input/output of real numbers on terminal or files. It works together with the module InOut. A description of this module is included to the Modula-2 manual [1].

Definition Modules

```

DEFINITION MODULE RealInOut; (* B.St. 26-May-86 *)

  PROCEDURE ReadReal(VAR x: REAL);
    (* read a real number from keyboard according to EBNF-syntax:
       ["-"|"+"]<digit>{<digit>}["."<digit>{<digit>}]
       ["E"|"-"|"+"<digit>{<digit>}] *)

  PROCEDURE WriteReal(x: REAL; n: CARDINAL);
    (* write a real number with n significant digits *)

END RealInOut.

```

```

DEFINITION MODULE LongRealInOut; (* B.St. 26-May-86 *)

  PROCEDURE ReadLongReal(VAR x: LONGREAL);
    (* read a real number from keyboard according to EBNF-syntax:
       ["-"|"+"]<digit>{<digit>}["."<digit>{<digit>}]
       ["E"|"-"|"+"<digit>{<digit>}] *)

  PROCEDURE WriteLongReal(x: LONGREAL; n: CARDINAL);
    (* write a real number with n significant digits *)

END LongRealInOut.

```

Imported Library Modules

```

InOut
RIO

```

7.4. MathLib, LongMathLib

Beat Stamm, 26-May-86

Library module providing some basic mathematical functions. A description of this module is included to the Modula-2 manual [1].

Definition Module

```
DEFINITION MODULE MathLib (* B.St. 26-May-86 *);
```

```
  PROCEDURE Sqrt  (x: REAL):    REAL;
  PROCEDURE Exp   (x: REAL):    REAL;
  PROCEDURE Ln    (x: REAL):    REAL;
  PROCEDURE Sin   (x: REAL):    REAL;
  PROCEDURE Cos   (x: REAL):    REAL;
  PROCEDURE Cos   (x: REAL):    REAL;
  PROCEDURE ArcTan(x: REAL):    REAL;
  PROCEDURE Real  (x: INTEGER):  REAL;
  PROCEDURE Entier(x: REAL):    INTEGER;
```

```
END MathLib.
```

```
DEFINITION MODULE LongMathLib (* B.St. 26-May-86 *);
```

```
  PROCEDURE LongSqrt (x: LONGREAL):  LONGREAL;
  PROCEDURE LongExp  (x: LONGREAL):  LONGREAL;
  PROCEDURE LongLn   (x: LONGREAL):  LONGREAL;
  PROCEDURE LongSin  (x: LONGREAL):  LONGREAL;
  PROCEDURE LongCos  (x: LONGREAL):  LONGREAL;
  PROCEDURE LongCos  (x: LONGREAL):  LONGREAL;
  PROCEDURE LongArcTan(x: LONGREAL):  LONGREAL;
  PROCEDURE LongReal (x: LONGINT):    LONGREAL;
  PROCEDURE LongEntier(x: LONGREAL):  LONGINT;
```

```
END LongMathLib.
```

Imported Library Modules

Terminal

7.5. FileSystem

Werner Heiz, 25-Feb-86

Library module providing basic sequential file access. A description of this module is included to the Modula-2 manual [1].

Definition Module

```
DEFINITION MODULE FileSystem; (* W. Heiz, 19-Dec-85 / 4-Feb-86 *)
```

```
FROM SYSTEM IMPORT WORD;
```

```
TYPE
```

```
  Response = (done, notdone);
```

```
  File      = RECORD
```

```
    refNum,
```

```
    volRef:      INTEGER;
```

```
    firstPos,
```

```
    lastPos,
```

```
    curPos:      LONGINT;
```

```
    res:         Response;
```

```
    eof:         BOOLEAN;
```

```
    dirty:       BOOLEAN;
```

```
    nameString:  ARRAY [0..31] OF CHAR;
```

```
    buffer:      ARRAY [0..1024-1] OF CHAR;
```

```
  END;
```

```
PROCEDURE Lookup(VAR f: File; filename: ARRAY OF CHAR; new: BOOLEAN);
```

```
PROCEDURE Close(VAR f: File);
```

```
PROCEDURE Delete(VAR f: File);
```

```
PROCEDURE Rename(VAR f: File; filename: ARRAY OF CHAR);
```

```
PROCEDURE SetPos(VAR f: File; highpos, lowpos: CARDINAL);
```

```
PROCEDURE GetPos(VAR f: File; VAR highpos, lowpos: CARDINAL);
```

```
PROCEDURE Length(VAR f: File; VAR highpos, lowpos: CARDINAL);
```

```
PROCEDURE ReadWord(VAR f: File; VAR w: WORD);
```

```
PROCEDURE WriteWord(VAR f: File; w: WORD);
```

```
PROCEDURE ReadChar(VAR f: File; VAR ch: CHAR);
```

```
PROCEDURE WriteChar(VAR f: File; ch: CHAR);
```

```
END FileSystem.
```

Explanations

Lookup(f, filename, new)

Procedure *Lookup* looks for the actual file with the given file name. If the file exists, it is connected to f (opened). If the requested file is not found or new is TRUE, a permanent file is created with the given name. After the call

f.res = done if file f is connected

f.res = notdone if the file does not exist or some error occurred

Close(f)

Procedure *Close* terminates any actual input or output operation on file f and disconnects the variable f from the actual file.

Delete(f)

Procedure *Delete* terminates any actual input or output operation on file *f* and disconnects the variable *f* from the actual file. The actual file is deleted hereafter.

Rename(f, filename)

Procedure *Rename* changes the name of file *f* to *filename*. After the call

f.res = done if file *f* is renamed

f.res = notdone if a file with *filename* already exists, or some error occurred

SetPos(f, highpos, lowpos)

A call to procedure *SetPos* sets the current position of file *f* to $\text{highpos} \cdot 2^{16} + \text{lowpos}$. The new position must be less or equal the length of the file.

GetPos(f, highpos, lowpos)

Procedure *GetPos* returns the current file position. It is equal to $\text{highpos} \cdot 2^{16} + \text{lowpos}$.

Length(f, highpos, lowpos)

Procedure *Length* gets the position just behind the last element of the file (i.e. the number of bytes stored on the file). The position is equal to $\text{highpos} \cdot 2^{16} + \text{lowpos}$.

WriteChar(f, ch), WriteWord(f, w)

Procedure *WriteChar* (*WriteWord*) appends character *ch* (word *w*) to file *f*.

ReadChar(f, ch), ReadWord(f, w)

Procedure *ReadChar* (*ReadWord*) reads the next character (word) from file *f* and assigns it to *ch* (*w*). If *ReadChar* has been called without success, 0C is assigned to *ch*. *f.eof* implies *ch* = 0C. The opposite, however, is not true: *ch* = 0C does not imply *f.eof*. After the call

f.eof = FALSE *ch* (*w*) has been read

f.eof = TRUE read operation was not successful

If *f.eof* is TRUE:

f.res = done end of file has been reached

f.res = notdone some error occurred

Examples

Writing a Text File

```

VAR
  f: File;
  ch: CHAR; endoftext: BOOLEAN;
...
Lookup(f, "newfile", TRUE);
IF f.res <> done THEN
  (* f was not created by this call to "Lookup" (disk full?) *)
ELSE
  LOOP
    (* find next character to write --> endoftext, ch *)
    IF endoftext THEN EXIT END;
    WriteChar(f, ch)
  END;
  Close(f);
  IF f.res <> done THEN
    (* f could not be closed correctly (disk full?) *)
  END;
END
...

```

Reading a Text File

```

VAR
  f: File;
  ch: CHAR;
...
Lookup(f, "oldfile", FALSE);
IF f.res <> done THEN
  (* file not found *)
ELSE
  LOOP
    ReadChar(f, ch);
    IF f.eof THEN EXIT END;
    (* use ch *)
  END;
  Close(f)
END
...

```

Imported Modules

Toolbox

7.6. Terminal, TerminalIn, TerminalOut

Christian Vetterli, 26-Feb-86

The module *TerminalIn* contains procedures to read characters from the keyboard.

Definition Module

```

DEFINITION MODULE TerminalIn;
  (* W. Heiz, 12-Dec-85 / C. Vetterli 26-Feb-86 *)

  PROCEDURE BusyRead(VAR ch: CHAR);
    (*Read a character from the keyboard and return it in ch if
      there is any, else return 0C*)

  PROCEDURE Read(VAR ch: CHAR);
    (*Read a character from the keyboard and return it in ch*)

END TerminalIn.

```

Explanations

Read(ch)

Procedure *Read* gets the next character from the keyboard (or from the commandfile) and assigns it to ch. The procedure *Read* does not "echo" the read character on the screen.

BusyRead(ch)

Procedure *BusyRead* assigns 0C to ch if no character has been typed. Otherwise *BusyRead* is identical to procedure *Read*.

By typing the command key and the key 'l' simultaneously you can reassign the input to a file (commandfile).

Imported Modules

EventBase
Toolbox

The module *TerminalOut* contains procedures to write characters to a standard window ("Terminal") on the screen.

Definition Module

```

DEFINITION MODULE TerminalOut;
  (* W. Heiz, 12-Dec-85 / C. Vetterli 26-Feb-86 *)

  PROCEDURE Write(ch: CHAR);
  PROCEDURE WriteString(string: ARRAY OF CHAR);
  PROCEDURE WriteLn;

END TerminalOut.

```

*Explanations***Write(ch)**

Procedure *Write* writes the given character on the screen at its current writing position. The screen scrolls, if the writing position reaches its end. Besides the following lay-out characters, it is left undefined what happens, if non printable ASCII characters and non ASCII characters are written out.

CR	15C	Sets the writing position at the beginning of the current line
LF	12C	Sets the writing position to the same column in the next line
BS	10C	Sets the writing position one character backward
DEL	177C	Sets the writing position one character backward and erases the character there

WriteString(string)

Procedure *WriteString* writes out the given string.

WriteLn

A call to procedure *WriteLn* is equivalent to the call *Write*(CR); *Write*(LF).

*Imported Modules***Toolbox**

The module *Terminal* contains all procedures of the modules *TerminalIn* and *TerminalOut*.

Definition Module

```

DEFINITION MODULE Terminal;
(* W. Heiz, 12-Dec-85 / C. Vetterli 26-Feb-86 *)

PROCEDURE BusyRead(VAR ch: CHAR);
PROCEDURE Read(VAR ch: CHAR);

PROCEDURE Write(ch: CHAR);
PROCEDURE WriteString(string: ARRAY OF CHAR);
PROCEDURE WriteLn;

END Terminal.
```

Imported Modules

TerminalIn
TerminalOut
Toolbox

7.7. Printer

Christian Vetterli, 15-Aug-86

The module *Printer* contains procedures to write characters to the printer.

Definition Module

```
DEFINITION MODULE Printer; (* C. Vetterli, 15-Aug-86 *)
```

```
  PROCEDURE Reset;  
  PROCEDURE Write(ch: CHAR);  
  PROCEDURE WriteString(str: ARRAY OF CHAR);  
  PROCEDURE WriteLn;
```

```
END Printer.
```

Imported Modules

Toolbox

7.8. CursorMouse

Christian Vetterli, 26-Feb-86

Module to request the mouse buttons, mouse position and to display a cursor on the screen.

Definition Module

```

DEFINITION MODULE CursorMouse; (* C. Vetterli, 4-Jan-86 / 26-Feb-86 *)

  CONST ML = 15; MM = 14; MR = 13;

  TYPE
    Pattern = RECORD
      raster: ARRAY[0..15] OF BITSET; (*see Inside Macintosh*)
      mask:   ARRAY[0..15] OF BITSET;
      hotSpv, hotSph: INTEGER
    END;

  PROCEDURE SetMouse(x, y: INTEGER);
    (* dummy Procedure; Macintosh handles cursor tracking *)

  PROCEDURE GetMouse(VAR s: BITSET; VAR x, y: INTEGER);
    (* Get current 'mouse' state
       ML IN s = "Mouse button pressed";
       MM IN s = "Command key pressed";
       MR IN s = "Option key pressed" *)

  PROCEDURE MoveCursor(x, y: INTEGER);
    (* Shows the cursor *)

  PROCEDURE EraseCursor;

  PROCEDURE SetPattern(VAR p: Pattern);
    (* Activate private cursor *)

  PROCEDURE ResetPattern;
    (* Reactivates standard arrow cursor *)

END CursorMouse.
```

Explanations

GetMouse returns the state of the buttons and the position of the mouse.

MoveCursor shows the cursor on the screen (parameters are only dummy).

EraseCursor deletes the cursor on the screen.

SetPattern is used to install a private cursor pattern. Subsequent calls of *MoveCursor* will use this pattern. The cursor need not to be erased before calling *SetPattern*.

ResetPattern returns to the default pattern (arrow) after *SetPattern* has been called.

Imported Modules

Toolbox

7.9. Menu

Christian Vetterli, 26-Feb-86

Module for command selection from a menu.

Definition Module

```
DEFINITION MODULE Menu; (* C. Vetterli, 4-Jan-86 / 26-Feb-86 *)
```

```
TYPE MenuRes = RECORD menuID, menuCmd: INTEGER END;
```

```
PROCEDURE SetMenu(menuID: INTEGER; menuStr: ARRAY OF CHAR);
(*installs a menu in the title bar
 menuID   = a user defined identification number.
 menuStr  = title {"|" item}.
 title, item = string. *)
```

```
PROCEDURE GetMenuCmd(VAR cmd: MenuRes; VAR done: BOOLEAN);
(*returns a menu command if done is TRUE*)
```

```
END Menu
```

Explanations

Procedure *SetMenu* installs a new menu at the end of the menu bar.

menuID is a user defined number. It is used to identify the activated menu command returned by *GetMenuCmd*.

menuStr contains all information to display a menu as defined in *Inside Macintosh* but as delimiter serves the character "|" !

menuCmd is the entry number which has been selected (first item is number 1).

Imported Modules

Toolbox

7.10. Windows

Christian Vetterli, 26-Feb-86

Library module for handling windows on the display. A description of a similar implementation on the Lilith computer is included in report Nr. 56 of the Institut für Informatik, ETH Zürich (Author: Jürg Gutknecht, Title: System Programming in Modula-2: Mouse and Bitmap Display).

Definition Module

```

DEFINITION MODULE Windows; (* C. Vetterli, 4-Jan-86 / 1-Jul-86 *)

  CONST Background = 0; FirstWindow = 1; LastWindow = 16;

  TYPE Window = [Background..LastWindow];
    (*Background serves as a possible return value for the
      UpWindow procedure and is not accessible to the user*)

  RestoreProc = PROCEDURE(Window);

  CloseProc = PROCEDURE(Window);

  ScrollDirection = (left, right, up, down, horizontal, vertical);

  ScrollProc = PROCEDURE(Window, ScrollDirection, INTEGER);

  PROCEDURE OpenWindow(VAR u: Window; x, y, w, h: INTEGER;
    repaint: RestoreProc; VAR done: BOOLEAN);
    (*Open a new window u and initialize its rectangle by clearing
      it and drawing a frame with a title bar; procedure repaint
      will be called when restoration becomes necessary*)

  PROCEDURE DrawTitle(u: Window; title: ARRAY OF CHAR);
    (*Draw title*)

  PROCEDURE RedefineWindow(u: Window; x, y, w, h: INTEGER;
    VAR done: BOOLEAN);
    (*Change and reinitialize rectangle of window u*)

  PROCEDURE CloseWindow(u: Window);

  PROCEDURE OnTop(u: Window): BOOLEAN;

  PROCEDURE PlaceOnTop(u: Window);

  PROCEDURE PlaceOnBottom(u: Window);

  PROCEDURE ScrollWindow(u: Window; dh, dv: INTEGER);
    (*Scrolls the content of the window by dh, dv*)

  PROCEDURE UpWindow(x, y: INTEGER): Window;
    (*Return the uppermost open window containing (x, y),
      if there is any, and take the value Background otherwise*)

```

```
PROCEDURE ActualSize(u: Window; VAR w, h: INTEGER);
  (*Return the actual size of the window*)

PROCEDURE SetWindow(u: Window);
  (*Sets the window u as current port*)

PROCEDURE ResetWindow;
  (*Sets the previous window as current port*)

PROCEDURE AssignScrollAction(u: Window; p: ScrollProc;
                             vert, horz: BOOLEAN);
  (*Installs scroll bar procedure and enables scroll bars*)

PROCEDURE SetScrollBars(u: Window; vert, horz: INTEGER);
  (*Positions sliders in scroll bars*)

PROCEDURE AssignCloseProc(u: Window; p: CloseProc);
  (*Installs close procedure and enables close box*)

PROCEDURE GetWindowFrame(u: Window; VAR x, y, w, h: INTEGER);
  (*Return the actual outline of the window*)

END Windows.
```

Imported Modules

```
EventBase
Toolbox
```

7.11. TextWindows

Hansrudolf Schär, 28-Oct-85, Christian Vetterli, 26-Feb-86

Library module for writing non-proportional text into a window. Text windows may be operated as scrolling displays or as forms (with positioning). A description of a similar implementation on the Lilith computer is included in report Nr. 56 of the Institut für Informatik, ETH Zürich (Author: Jürg Gutknecht, Title: System Programming in Modula-2: Mouse and Bitmap Display).

Definition Module

```
DEFINITION MODULE TextWindows; (* C. Vetterli, 4-Jan-86 / 1-Jul-86 *)
```

```
  IMPORT Windows;
```

```
  VAR Done: BOOLEAN;
      (* Done = "previous operation was successfully executed" *)
      termCH: CHAR; (* termination character *)
```

```
  TYPE Window = Windows.Window;
      RestoreProc = Windows.RestoreProc;
      CloseProc = Windows.CloseProc;
      ScrollDirection = Windows.ScrollDirection;
      ScrollProc = Windows.ScrollProc;
```

```
  PROCEDURE OpenTextWindow(VAR u: Window; x,y,w,h: INTEGER;
                           name: ARRAY OF CHAR);
```

```
    (*Open new standard text window. Draw title bar*)
```

```
  PROCEDURE RedefTextWindow(u: Window; x,y,w,h: INTEGER);
```

```
    (*Redefine and reinitialize window u*)
```

```
  PROCEDURE CloseTextWindow(u: Window);
```

```
  PROCEDURE AssignFont(u: Window; fontNum, size: INTEGER);
    (*Assign non-proportional font*)
```

```
  PROCEDURE TextWindowSize(u: Window; VAR lines, col: INTEGER);
    (*Return the actual size of the window*)
```

```
  PROCEDURE AssignRestoreProc(u: Window; r: RestoreProc);
    (*Assign procedure to restore window u*)
```

```
  PROCEDURE AssignEOWAction(u: Window; r: RestoreProc);
    (*Assign reaction on "end of window" condition*)
```

```
  PROCEDURE ScrollUp(u: Window);
    (*Scroll one line up*)
```

```
  PROCEDURE DrawTitle(u: Window; name: ARRAY OF CHAR);
```

```
  PROCEDURE DrawLine(u: Window; line, col: INTEGER);
```

```
    (*col = 0: draw horizontal line at line;
```

```
    line = 0: draw vertical line at col *)
```

```
  PROCEDURE SetCaret(u: Window; on: BOOLEAN);
```

```
    (*on TRUE (FALSE): turn caret on (off)*)
```

```
  PROCEDURE Invert(u: Window; on: BOOLEAN);
```

```
    (*on TRUE (FALSE): set to inverse video (normal) mode*)
```

```

PROCEDURE IdentifyPos(u: Window; x, y: INTEGER;
                     VAR line, col: INTEGER);
  (*Identify window, line and col corresponding to screen
    coordinates (x,y)*)
PROCEDURE GetPos(u: Window; VAR line, col: INTEGER);
  (*Get current position*)
PROCEDURE SetPos(u: Window; line, col: INTEGER);
  (*Set within window u*)
PROCEDURE ReadString(u: Window; VAR a: ARRAY OF CHAR);
  (*Read a String and echo to window u*)
PROCEDURE ReadCard(u: Window; VAR x: CARDINAL);
  (*Read a Cardinal and echo to window u;
    leading blanks are ignored*)
PROCEDURE ReadInt(u: Window; VAR x: INTEGER);

PROCEDURE Write(u: Window; ch: CHAR);
  (* Write character ch at current position.
    BS, LF, FF, CR, CAN, EOL and DEL are interpreted *)
PROCEDURE WriteLn(u: Window);
PROCEDURE WriteString(u: Window; a: ARRAY OF CHAR);
PROCEDURE WriteCard(u: Window; x, n: CARDINAL);
  (* Write integer x with (at least) n characters.
    If n is greater than the number of digits needed,
    blanks are added preceding the number *)
PROCEDURE WriteInt(u: Window; x, n: INTEGER);
PROCEDURE WriteOct(u: Window; x, n: CARDINAL);

PROCEDURE AssignScrollAction(u: Window; p: ScrollProc;
                             vert, horiz: BOOLEAN);
PROCEDURE SetScrollBars(u: Window; vert, horiz: INTEGER);
PROCEDURE AssignCloseProc(u: Window; p: CloseProc);

```

END TextWindows.

Imported Modules

Windows
 EventBase
 Toolbox

7.12. GraphicWindows

Eliyezer Kohen, Beat Stamm, 26-May-86

Library module for drawing graphics into a window.

Definition Module

```

DEFINITION MODULE GraphicWindows; (* E. Kohen *)

  IMPORT Windows;

  TYPE Window = Windows.Window;
      RestoreProc = Windows.RestoreProc;
      Mode = (replace, paint, invert, erase);

  VAR Done: BOOLEAN; (* Done = "Operation was successfully executed" *)

  PROCEDURE OpenGraphicWindow(VAR u: Window; x,y,w,h: INTEGER;
      name: ARRAY OF CHAR; Repaint: RestoreProc);
    (* Open new graphic window. Draw title bar if "name" not empty *)

  PROCEDURE RedefGraphicWindow(u: Window; x,y,w,h: INTEGER);
    (* Change rectangle and reinitialize graphic window u *)

  PROCEDURE Clear(u: Window);
    (* Clear window u *)

  PROCEDURE CloseGraphicWindow(u: Window);
    (* Close window u *)

  PROCEDURE SetMode(u: Window; m: Mode);
    (* Set mode *)

  PROCEDURE Dot(u: Window; x, y: INTEGER);
    (* Place dot at coordinate x,y *)

  PROCEDURE SetPen(u: Window; x, y: INTEGER);
    (* Set pen at (x,y) *)

  PROCEDURE TurnTo(u: Window; d: INTEGER);
    (* Turn pen direction to d degrees *)

  PROCEDURE Turn(u: Window; d: INTEGER);
    (* Turn pen direction by d degrees *)

  PROCEDURE Move(u: Window; n: INTEGER);
    (* Move pen and draw line of length n in specified direction *)

  PROCEDURE MoveTo(u: Window; x, y: INTEGER);
    (* Move pen and draw line to (x,y) *)

  PROCEDURE Circle(u: Window; x,y, r: INTEGER);
    (* Draw circle with center at (x,y) and radius r *)

```

```
PROCEDURE Area(u: Window; c: INTEGER; x,y,w,h: INTEGER);  
  (* Paint rectangular area of width w and height h at (x,y)  
    in color c: 0: white, 1: light grey, 2: dark grey, 3: black *)
```

```
PROCEDURE CopyArea(u: Window; sx,sy,dx,dy,dw,dh: INTEGER);  
  (* Copy rectangular area at (sx,sy) into rectangle at (dx,dy)  
    of width dw and height dh *)
```

```
PROCEDURE Write(u: Window; ch: CHAR);  
  (* Write ch at pen position *)
```

```
PROCEDURE WriteString(u: Window; s: ARRAY OF CHAR);  
  (* Write string s at pen position *)
```

```
PROCEDURE IdentifyPos(VAR u: Window; VAR x,y: INTEGER);  
  (* Return window and relative coordinates corresponding to  
    screen coordinates (x,y) *)
```

END GraphicWindows.

Imported Modules

Windows
EventBase
MathLib
Toolbox

7.13. EventBase

Christian Vetterli, 26-Feb-86

Module *EventBase* provides a simple scheduler intended to allow arranging a Modula-2 program on the Macintosh as a collection of event handling subroutines. This module is conceptually built on top of the Macintosh *Toolbox Event Manager* and its main goal is to support the decomposition of an otherwise tremendously large CASE-statement for event handling into independent modules.

"A typical Macintosh application program is event-driven: It decides what to do from moment to moment by asking the Event Manager for events and responding to them one by one in whatever way is appropriate." (*Inside Macintosh, About the Toolbox Event Manager*)

DEFINITION MODULE EventBase; (* C. Vetterli, 4-Jan-85 / 26-Feb-85 *)

CONST

```
(* events *)
nullEvent = 0; mouseDown = 1; mouseUp = 2; keyDown = 3;
keyUp = 4; autoKey = 5; updateEvt = 6; diskEvt = 7;
activateEvt = 8; networkEvt = 10; driverEvt = 11; app1Evt = 12;
app2Evt = 13; app3Evt = 14; app4Evt = 15;
```

TYPE

```
Point = RECORD v, h: INTEGER END;
EventRecord = RECORD
    what: INTEGER; (* event code *)
    message: LONGINT; (* event message *)
    when: LONGINT; (* ticks since startup *)
    where: Point; (* mouse location *)
    modifiers: INTEGER; (* modifier flags *)
END;
```

```
EventHandler = PROCEDURE(VAR EventRecord): BOOLEAN;
    (* this procedure type is used by the scheduler for *)
    (* distributing events obtained by GetNextEvent *)
    (* to the activated tasks *)
```

```
PROCEDURE PushTask(EventProc: EventHandler): INTEGER;
    (* push Task onto the scheduler and activate it *)
    (* returns task number *)
```

```
PROCEDURE PopTask(taskNum: INTEGER);
    (* pop a Task from the scheduler and deactivate it *)
```

```
PROCEDURE PopallTasks;
    (* deactivate and remove all entries in the Task scheduler *)
```

```
PROCEDURE PollEventTasks;
    (* to be called repeatedly in the Main Loop of a Program *)
```

```

PROCEDURE GetBusyReadEvent(VAR event: EventRecord): BOOLEAN;
(* get keyDown and autoKey events that no active Task wanted
   to handle *)
(* this enables the Module Terminal to run as lowest priority
   handler *)

```

END EventBase.

Explanations

EventHandler

This procedure is used by the scheduler for distributing events obtained by *GetNextEvent (Inside Macintosh)* to the activated tasks. The *EventHandler* first has to analyse the event, then either handle it and return TRUE or not handle it and return FALSE.

PollEventTasks

This procedure is to be called repeatedly in the main loop of a program. *PollEventTasks* performs a *GetNextEvent*, distributes events to the activated *EventHandler* and calls the Toolbox procedure *SystemTask* if no events are pending.

GetBusyReadEvent

KeyDown or *AutoKey* events that remain unhandled by the tasks are buffered to be used by module *TerminalIn*.

Imported Modules

Toolbox

7.14. System

Christian Vetterli, Hermann Seiler, 26-Feb-86

The module *System* is the heart of the Modula-2 system on the Macintosh. It contains the loader and procedures to supply missing instructions of the processor (REAL and LONGINT arithmetic). There are also procedures for calling and terminating programs and handling the heap.

Definition Module

DEFINITION MODULE System;

(* H. Seiler, C. Vetterli, 22-Dec-85 / 26-Feb-86 *)

FROM SYSTEM IMPORT ADDRESS;

(* 'System' FOR MC68000/MC68010 runtime support of Modula-2. *)

```

PROCEDURE HALTX;      (* argument in register D0 ! *)

PROCEDURE MULU32;      (* arguments and quadword-result in regs. D0/D1 *)
PROCEDURE DIVU32;      (* arguments and quadword-result in regs. D0/D1 *)
PROCEDURE MULS32;      (* arguments and quadword-result in regs. D0/D1 *)
PROCEDURE DIVS32;      (* arguments and quadword-result in regs. D0/D1 *)

PROCEDURE FADDs        (addend, addend : REAL)                : REAL;
PROCEDURE FSUBs        (minuend, subtrahend : REAL)           : REAL;
PROCEDURE FMULs        (multiplicand, multiplier : REAL)      : REAL;
PROCEDURE FDIVs        (dividend, divisor : REAL)             : REAL;
PROCEDURE FREMs        (dividend, divisor : REAL)             : REAL;
PROCEDURE FCMPs        (first, second : REAL);                (* result in CCR *)
PROCEDURE FNEGs        (toNeg : REAL)                          : REAL;
PROCEDURE FABs        (toAbs : REAL)                           : REAL;
PROCEDURE FLOATs       (toFloat : LONGINT)                     : REAL;
PROCEDURE TRUNCs       (toTrunc : REAL)                         : LONGINT;

PROCEDURE FADDd        (addend, addend : LONGREAL)            : LONGREAL;
PROCEDURE FSUBd        (minuend, subtrahend : LONGREAL)       : LONGREAL;
PROCEDURE FMULD        (multiplicand, multiplier : LONGREAL)  : LONGREAL;
PROCEDURE FDIVd        (dividend, divisor : LONGREAL)         : LONGREAL;
PROCEDURE FREMd        (dividend, divisor : LONGREAL)         : LONGREAL;
PROCEDURE FCMPd        (first, second : LONGREAL);             (* result in CCR *)
PROCEDURE FNEGd        (toNeg : LONGREAL)                      : LONGREAL;
PROCEDURE FABd        (toAbs : LONGREAL)                       : LONGREAL;
PROCEDURE FLOATd       (toFloat : LONGINT)                     : LONGREAL;
PROCEDURE TRUNCd       (toTrunc : LONGREAL)                    : LONGINT;

PROCEDURE FLONG        (toConvert : REAL)                      : LONGREAL;
PROCEDURE FSHORT       (toConvert : LONGREAL)                  : REAL;

```

CONST MaxPrograms = 6;

TYPE Status = (normal, moduleNotFound, fileNotFound, illegalKey,
readError, badSyntax, noMemory, alreadyLoaded, killed,
tooManyPrograms, continue, noApplication);

```

VAR A4, initAdr: LONGINT;
(* these two variables have to be declared first in this module *)
interruptPending: BOOLEAN;
(* signals a pending non maskable interrupt *)

PROCEDURE Allocate(VAR ptr: ADDRESS; size: LONGINT);
PROCEDURE Deallocate(VAR ptr: ADDRESS);

PROCEDURE Call(module: ARRAY OF CHAR; leaveLoaded: BOOLEAN;
               VAR status: Status);
PROCEDURE Terminate(status: Status);
PROCEDURE DisplayStatus(status: Status);

PROCEDURE InitProcedure(init: PROC; VAR done: BOOLEAN);
PROCEDURE TermProcedure(term: PROC; VAR done: BOOLEAN);

PROCEDURE GetProcessStatus(VAR err: ARRAY OF CHAR);
PROCEDURE GetNextModule(VAR modNo: INTEGER; VAR dataBase: LONGINT;
                       VAR modName: ARRAY OF CHAR);
PROCEDURE GetNextProcedure(VAR modNo: INTEGER; VAR dataBase: LONGINT;
                          VAR relPC: INTEGER);

(* HFS Support: *)

TYPE Path = ARRAY [0..63] OF CHAR;

PROCEDURE FindPath(VAR p: Path; in: ARRAY OF CHAR; VAR n: INTEGER);
(* find n-th path to a given filename *)
PROCEDURE AddPath (p: Path; in: ARRAY OF CHAR;
                  VAR out: ARRAY OF CHAR);
(* add path p to a given filename *)

END System.

```

Explanations

Allocate(a, size)

Procedure *Allocate* tries to allocate a memory area of the given size in the heap. If the space is not available, *a* returns NIL otherwise it returns the address of the reserved area.

Deallocate(a)

Procedure *Deallocate* releases the memory area given by address *a*. *a* returns the value NIL.

Implementation Note

The minimum memory area that will be reserved by *Allocate* is size+4 bytes. Only the heap of the program that runs on the topmost level should be expanded. All other heaps *must not* allocate any memory through *Allocate*.

Call(module, leaveLoaded, status)

Procedure *Call* activates a new program that executes the body of module *module*. The current program level is incremented by one before the execution of the program. If *leaveLoaded* is TRUE, all modules imported directly or indirectly by the

program are marked as non removable and will stay in memory for the lifetime of the application, otherwise the new loaded modules are removed after program termination. After the execution of the program, *status* indicates the termination cause.

Terminate(*status*)

The currently running process may be terminated by a call to *Terminate*. *status* may signal the cause of termination.

DisplayStatus(*status*)

Shows the *status* in a dialog box.

InitProcedure(*init*, *done*)

A call to *InitProcedure* installs an initialization procedure *init* which will be called whenever a program on a higher level is started. These initialization procedures are called just after the installation of the new execution level in question (i.e. after the current level is incremented) and in order of their announcements. The result parameter *done* is set TRUE if the assignment has been successful.

TermProcedure(*term*, *done*)

A call to *TermProcedure* installs a termination procedure *term* which will be called whenever a program on a higher level is terminated. These termination procedures are called just before the termination of the execution level in question (i.e. before the current level is reset) and in reverse order of their announcements. The result parameter *done* is set TRUE if the assignment has been successful.

(1321) FindPath(*p*, *in*, *n*)

The *n*th path for a given file name is returned in *p*. If the file name already contains a path, this path is returned. If there are no more paths, *n* is set to 0.

AddPath(*p*, *in*, *out*)

The path *p* and the file name *in* are combined to the full file name *out*.

Implementation Note

The total number of installed initialization and termination procedures is limited to 16 each. The procedures *GetProcessStatus*, *GetNextModule* and *GetNextProcedure* are used by the debugger exclusively.

7.15. Accessing the Macintosh Toolbox

Werner Heiz, 7-Apr-86

This chapter is intended for advanced programmers only, being familiar with the 1000-page manual *Inside Macintosh* [2].

It is possible to call each Toolbox, QuickDraw, and Operating System routine. The following example shows the way it is done:

```

MODULE ToolboxDemo; (* W. Heiz, 7-Apr-86 *)

FROM SYSTEM IMPORT
  ADDRESS, BYTE, WORD, SETREG, REG, INLINE;

TYPE
  Ptr      = ADDRESS;
  Handle   = POINTER TO ADDRESS;
  Str255   = ARRAY [0..255] OF CHAR;
  VHSelect = (V, H);
  Point    = RECORD
    CASE :INTEGER OF
      0: v, h: INTEGER;
      | 1: vh:  ARRAY [V..H] OF INTEGER;
    END;
  END;
  Rect     = RECORD
    CASE :INTEGER OF
      0: top, left, bottom, right: INTEGER;
      | 1: topLeft, botRight:      Point;
    END;
  END;
  BitMap   = RECORD
    baseAddr: Ptr;
    rowBytes: INTEGER;
    bounds:   Rect;
  END;
  Pattern  = ARRAY [0..7] OF BYTE;
  StyleItem = (bold, italic, underline, outline, shadow,
    condense, extend);
  Style     = SET OF StyleItem;
  Bits16    = ARRAY [0..15] OF CARDINAL;
  Cursor    = RECORD data, mask: Bits16; hotSpot: Point END;
  GrafPtr   = POINTER TO GrafPort;
  GrafPort  = RECORD
    device:          INTEGER;
    portBits:        BitMap;
    portRect:        Rect;
    visRgn, clipRgn: Handle;
    bkPat, fillPat:   Pattern;
    pnLoc, pnSize:    Point;
    pnMode:           INTEGER;
    pnPat:            Pattern;
    pnVis, txFont:    INTEGER;
    txFace:           Style;
    txMode, txSize:   INTEGER;
    spExtra, fgColor, bkColor: LONGINT;
  END;

```

```

        colrBit, patStretch:      INTEGER;
        picSave, rgnSave, polySave: Handle;
        grafProcs:                Ptr;
    END;

PROCEDURE GetPort(VAR port: GrafPtr); CODE 0A874H;
PROCEDURE SetPort(newPort: GrafPtr); CODE 0A873H;
PROCEDURE SetOrigin(h, v: INTEGER); CODE 0A878H;
PROCEDURE Random(): INTEGER; CODE 0A861H;
PROCEDURE SetRect(VAR r: Rect; left, top, right, bottom: INTEGER);
    CODE 0A8A7H;
PROCEDURE Pt2Rect(pt1, pt2: Point; VAR rect: Rect); CODE 0A8ACH;
PROCEDURE OffsetRect(VAR r: Rect; dh, dv: INTEGER); CODE 0A8A8H;
PROCEDURE InsetRect(VAR r: Rect; dh, dv: INTEGER); CODE 0A8A9H;
PROCEDURE PaintRect(VAR r: Rect); CODE 0A8A2H;
PROCEDURE EraseRect(VAR r: Rect); CODE 0A8A3H;
PROCEDURE InvertRect(VAR r: Rect); CODE 0A8A4H;
PROCEDURE DrawChar(ch: WORD); CODE 0A883H;
PROCEDURE NewWindow(wStorage: Ptr;
    VAR boundsRect: Rect;
    VAR title: Str255;
    visible: BOOLEAN;
    theProc: INTEGER;
    behind: Ptr;
    goAwayFlag: BOOLEAN;
    refCon: LONGINT): Ptr; CODE 0A913H;
PROCEDURE DisposeWindow(theWindow: Ptr); CODE 0A914H;

PROCEDURE NewPtr(byteCount: LONGINT): ADDRESS;
    CONST D0 = 0; A0 = 8;
    BEGIN
        SETREG(D0, byteCount); INLINE(0A11EH); RETURN REG(A0)
    END NewPtr;

PROCEDURE ModulaToPascalString(m: ARRAY OF CHAR; VAR p: Str255);
    VAR i: INTEGER;
    BEGIN
        i := 0;
        WHILE (i <= HIGH(m)) & (m[i] # 0C) DO p[i+1] := m[i]; INC(i) END;
        p[0] := CHR(i);
    END ModulaToPascalString;

PROCEDURE CalledByToolbox;
    BEGIN
        INLINE(048E7H, 01F38H); (* MOVEM.L A2-A4/D3-D7, -(SP) save regs *)
        (* your statements *)
        INLINE(04CDFH, 01CF8H); (* MOVEM.L (SP)+, A2-A4/D3-D7 rest regs *)
    END CalledByToolbox;

END ToolboxDemo.

```

Remarks:

1. Use same constant-, type-, variable-, procedure-, and function-declarations as in Lisa Pascal (but obey the Modula-2 syntax).
2. Declare parameters with size > 4 bytes as VAR parameters (the Macintosh expects

an address for these parameters).

3. Use the reserved word `CODE` to specify the trap number. Code procedures may also be placed within definition modules.
4. Replace type `CHAR` by type `WORD` (the Macintosh uses a 2-byte representation for type `CHAR`, Modula-2 uses only one byte).
5. Pascal and Modula-2 strings differ in their representation. You have to do the conversion explicitly before calling a Toolbox routine (see procedure `ModulaToPascalString`).
6. Pascal and Modula-2 use the same `NIL` representation (0D).
7. Modula-2 procedures may be passed as parameters to Toolbox routines if they save registers A2-A4/D3-D7 (see procedure `CalledByToolbox`).
8. Use `INLINE`, `SETREG`, and `REG` to access operating system routines (see procedure `NewPtr`). Register A5 is never touched by the compiler, registers D0 and D1 only in special cases (see module `System`).

theor. ab

theor. ab

theor. ab

theor. ab

theor. ab

theor. ab

theor. ab

theor. ab

